Data access performance impact of a memory-centric radio network

Tero Lindfors

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 18.09.2018

Supervisor

Prof. Tarik Taleb

Advisor

M.Sc. (Tech.) Jari Karppinen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Tero Lindfors

Työn nimi Muistikeskeisen radioverkon vaikutus tietopääsyjen suoritusnopeuteen

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Tietoliikennetekniikka **Pääaineen koodi** ELEC3029

Työn valvoja Prof. Tarik Taleb

Työn ohjaaja M.Sc. (Tech.) Jari Karppinen

Päivämäärä 18.09.2018 **Sivumäärä** 67+2 **Kieli** Englanti

Tiivistelmä

Tulevaisuuden 5G:hen perustuvissa mobiiliverkoissa verkkolaitteisto on pääosin virtualisoitu. Tällaisen verkon virtuaaliverkkolaite (*VNF*) koostuu ohjelmistokomponenteista, joita ajetaan tarkoitukseen määrätyiltä mobiiliverkon pilven palvelimilta. Ohjelmistokomponentti voi olla ajossa millä vain mobiiliverkon näistä pilvi-infrastruktuurin palvelimista. Palvelimet on tavallisesti yhdistetty TCP/IP-verkolla.

Jotta suunnitellut alhaisen viiveen käyttötapaukset voisivat toteutua 5G-verkoissa, pilvipalvelimia on sijoitettu niin kutsuttuihin reunadatakeskuksiin lähelle loppukäyttäjiä. Monet näistä käyttötapauksista sisältävät tietopääsyjä virtuaaliverkkolaitteesta toisiin tai muihin verkkoelementteihin. Tietopääsyviiveen halutaan olevan mahdollisimman pieni, jotta käyttötapausten tiukoissa viiverajoissa pysytään. Mahdollisena lähestymistapana tietopääsyviiveen minimoimiseen tutkittiin *muistikeskeistä* laitteistoalustaa. Tämän laitteistoalustan pääperiaatteita on korvata nykyiset lyhytkestoiset ja pysyvät muistit haihtumattomalla muistilla sekä kommunikoida muistisemanttisilla viesteillä tietokonekomponenttien kesken.

Tässä työssä muistikeskeisyyttä hyödyntävää sijaislaitteistoa käytettiin VNF-datan varastona ja ohjelmistokomponenttien tietopääsyviivettä sinne mitattiin erilaisilla kokeilla. Kokeet osoittivat nykyisen, TCP/IP-pohjaisen alustan huomattavasti muistikeskeistä alustaa hitaammaksi. Toiseksi, kokeet osoittivat tietopääsyviiveiden olevan saman suuruisia muistikeskeisen alustan sisällä, riippumatta saatavilla olevasta muistikaistasta. Tulokset merkitsevät, että muistikeskeinen alusta on lupaava vaihtoehto reunadatakeskuksen tietovarastojärjestelmäksi. Lisää tutkimusta alustasta kuitenkin tarvitaan, jotta muiden palvelun laatukriteerien, kuten matalan viivevaihtelun, toteutumisesta saadaan tietoa.

Avainsanat muistikeskeinen, virtuaaliverkkolaite, Gen-Z, muistisemanttinen, SHM, tietovarastoalusta, pääsyviive

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Tero Lindfors

Title Data access performance impact of a memory-centric radio network

Degree programme Computer, Communication and Information Sciences

Major Communications Engineering **Code of major** ELEC3029

Supervisor Prof. Tarik Taleb

Advisor M.Sc. (Tech.) Jari Karppinen

Date 18.09.2018 **Number of pages** 67+2 **Language** English

Abstract

Future 5G-based mobile networks will be largely defined by virtualized network functions (VNF). The related computing is being moved to cloud where a set of servers is provided to run all the software components of the VNFs. Such software component can be run on any server in the mobile network cloud infrastructure. The servers conventionally communicate via TCP/IP -network.

To realize planned low-latency use cases in 5G, some servers are placed to data centers near the end users (edge clouds). Many of these use cases involve data accesses from one VNF to another, or to other network elements. The accesses are desired to take as little time as possible to stay within the stringent latency requirements of the new use cases. As a possible approach for reaching this, a novel memory-centric platform was studied. The main ideas of the memory-centric platform are to collapse the hierarchy between volatile and persistent memory by utilizing non-volatile memory (NVM) and use memory-semantic communication between computer components.

In this work, a surrogate memory-centric platform was set up as a storage for VNFs and the latency of data accesses from VNF application was measured in different experiments. Measurements against a current platform showed that memory-centric platform was significantly faster to access than the current, TCP/IP using platform. Measurements for accessing RAM with different memory bandwidths within the memory-centric platform showed that the order of latency was roughly independent of the available memory bandwidth. These results mean that memory-centric platform is a promising alternative to be used as a storage system for edge clouds. However, more research is needed to study how other service qualities, such as low latency variation, are fulfilled in memory-centric platform in a VNF environment.

Keywords memory-centric, virtualized network function, Gen-Z, memory-semantic, SHM, storage back-end, access latency

Acknowledgements

This thesis was made at Nokia Oyj in Espoo, Finland. I would like to thank my instructor Jari Karppinen and supervisor Tarik Taleb for helpful feedback and guidance during the study. Additionally, I would like to thank my colleagues from Nokia, especially Erkki Hietala, Arvo Heinonen and Markus Teivo. They offered me wonderful support for finding the most applicable experimental methods and implementation practices for the thesis research. Finally, I would like to acknowledge my friends, family and girlfriend for a good support throughout the entire thesis project.

Espoo, 18.9.2018

Tero T. Lindfors

Contents

Abbreviations

5G	Fifth Generation Mobile Network
NFV	Network Function Virtualization
VNF	Virtualized Network Function
SDL	Shared Data Layer
SOC	System On A Chip
ETSI	European Telecommunications Standards Institute
NFVI	Network Functions Virtualisation Infrastructure
VM	Virtual Machine
COTS	Commercial-Off-The-Shelf
VNFC	Virtualized Network Function Component
MME	Mobility Management Entity
NUMA	Non-Uniform Memory Access
PCN	Processor-Centric Network
IP	Internet Protocol
TCP	Transmission Control Protocol
NVM	Non-Volatile Memory
RAM	Random-Access Memory
DRAM	Dynamic Random-Access Memory
I/O	Input/Output
MCN	Memory-Centric Network
ToR	Top-Of-The-Rack
CPU	Central Processing Unit
OS	Operating System
SSD	Solid-State Drive
HDD	Hard Disk Drive
FPGA	Field-Programmable Gate Array
MMU	Memory Management Unit
WDM	Wavelength Division Multiplexing
HMC	Hybrid Memory Cube
3GPP	Third Generation Partnership Project
KVM	Kernel-based Virtual Machine
API	Application Programming Interface
FAM	Fabric-Attached Memory
NVRAM	Non-Volatile Random-Access Memory
DBMS	Database Management System
NoSQL	Non-Structured Query Language
FAME	Fabric Attached Memory Emulator
DDR4	Double Data Rate Fourth Generation
ASIC	Application-Specific Integrated Circuit
SHM	Linux Shared Memory
GNU	Gnu's Not Unix

1 Introduction

Sharing of rich multimedia content via mobile devices has continued to grow fast in recent years [1, p. 17]. The evident Internet of Things (IoT) appliances will make impersonal machines to join the network as well [1, p. 18]. Such use cases will be supported in next mobile technology generation, referred as the fifth generation (5G), which requires high bandwidth and ultra-low latency services from the novel networks [2]. To provide such services, the performance and versatility of a network should be improved from the level offered by current networks. While network operators need infrastructure and systems capable of fulfilling the performance demands, the incurring costs should not lead to increases in subscription fees [3]. ETSI standardization institute has developed a Network Function Virtualization (NFV) design to address these challenges from the point of view of both network operators and engineers. The essence of NFV design is to provide network functions as pure software instead of conventional dedicated components [3]. This enables to build a network of commodity hardware and change the resource allocation of each network function [3]. Utilizing a pool of hardware in such a way is referred as cloud computing. To realize these characteristics in 5G networks, the network architecture requires careful redesigning and many players on the field have already published blueprints about their approaches [3][2].

The performance of an NFV mobile network depends on how well its cloud resources can process the traffic generated by all the connected equipment. One important part of the processing is accessing data from storage locations in the network. NFV applications use data for various purposes, such as session or subscriber management [4]. To prevent performance declining, the accessing should be able to follow the rates at which network function software processes the traffic. Furthermore, certain services require other properties from the storage system to ensure quality. Conventional storage systems have been showed to have difficulties in adapting to these needs. Recently, a storage system with a totally different interconnect architecture than in current systems has been presented [5][6]. This *memory-centric network* (MCN) architecture arranges a part of every compute nodes' memory into a shared resource where each memory module is interconnected by optics, forming a memory fabric [6]. Due to this, access to any memory location in the total memory of storage elements becomes equivalent and causes roughly equal latency. [6] One major initiative studying applications of this type of architecture is The Machine -program, led by Hewlett Packard Enterprise [7].

The MCN architecture has shown promising performance in several application areas, which makes it worth exploring as a storage system for 5G network. In Nokia's 5G network architecture blueprint, *cloud-native core network* concept encompasses the cloud resources used for computing [4]. Storage resources are comprised in a special *Shared Data Layer* (SDL) [8]. Virtualized network functions (VNF) access the storage via SDL, which reflects the performance of accessing the underlying storage [8]. Besides high performance, 5G mobile network is expected to guarantee service availability and reliability and in some cases consistent shared memory accesses. Suitability of the MCN architecture is considered from the point of view of these

qualities as well.

To get closer to a mobile network that is capable to serve the new use cases, network engineers aim to reduce data access latency in VNF software [9]. The engineers and network operators are highly interested of the effects on access latency of VNF applications that a changed interconnect architecture would yield. The aim of this work was to study how this access latency could be quantified and apply the discovered method to compare performance of a conventional TCP/IP -based platform and a MCN platform. Access latency was used as the performance metric. As deployable storage systems that followed the novel interconnect architecture completely did not exist by the time of the work, memory-centric storage system was simulated with a surrogate platform, called *Superdome Flex*. Superdome Flex utilizes some of the technologies which are employed in The Machine -prototype [7][10]. Accessing of data from this platform was made via a database based on conventional Linux shared memory [11, p. 997][7]. However, the study seeked to get a proof of concept type of results to see differences in performance between current, TCP/IP -based systems and memory-centric systems.

2 Related design and technologies

To understand the environment in which mobile network services are operated, the most important design and technology are explained and discussed. The topics are emphasized on areas which effect on the data access performance of network function components. They provide necessary background to understand novel mobile network design and compare it to the existing design on data utilities. Cloud computing is a central technique to enable the new design.

2.1 Design

To provide network services, the radio network infrastructure and the systems built on top of them need appropriate design. As services have begun to require lots of computing power, cloud hardware is often utilized as part of the infrastructure. This constituent is referred as telecommunication (telco) cloud. The computing units of a telco cloud have typically a common shared memory to share data over the software components of the network. As the most feasible cloud solutions are built of multiple nodes, networking performance in the node level remains important in the near future design. This makes communication and arrangement of nodes significant: these are discussed in "System interconnect architectures" and "Topologies and routing" sections. Virtualization and standardization can improve efficiency and scalability of a network and they have effects on the expected service characteristics. "Virtualized Network Function" and "Quality characteristic of cloud storage" review these two topics.

2.1.1 General cloud service composition

Present-day cloud service is built of a set of servers, networking elements, storage elements and the software allowing of the supply of these resources [12]. These mentioned hardware and software are commonly termed as cloud infrastructure. A service model that solely supplies cloud infrastructure to consumers is called Infrastructure as a Service (IaaS) [12]. An individual server is called a node. Nodes and storage elements are connected via network, which altogether form a cluster [3, p. 236]. The usual installation site for cloud infrastructure is a data center where clusters are individually packed into racks and connected to each other via upper level networking devices [13, pp. 219-220].

In telco clouds, a cluster is commonly a distributed system where the nodes are further linked by middleware software to coordinate their actions and resource sharing. The distributed system is so perceived as a single computing facility by consumers. [13, p. 27]. On telco applications, a single node of a cluster is typically a computer based on System On A Chip (SOC) architecture and the hardware resources are usually virtualized. Both the physical hardware and the virtualized resources are part of IaaS which can be mapped to the bottom level of telco cloud architecture. [3, p. 243].

The foreseeable mobile computing use cases require higher degree of responsiveness, flexibility, scalability and performance from the mobile network [4]. All these

properties entail lots of computing power and dynamicity from the network, which makes redesigning of the existing networks crucial [1, pp. 131, 173]. Smart network of data centers can provide the needed resources [4]. Placing small-scale data centers near the end users where data is accessed allows sufficient supply of resources [4]. Such data centers are called edge clouds, and a bunch of such are coordinated in a distributed manner to make full use of resources for requests [4]. In addition to edge cloud, centralized data centers can be utilized to manage the overall resource delivery and serve as origin nodes [1, p. 176].

The dynamicity can be achieved by transforming the network software into *microservices* architecture [4]. Microservices split the current, standalone network function applications into smaller software modules, defined as services [4]. A service is typically a set of processes fulfilling a specific business duty, including a means to communicate with other services [14]. The microservice transformation leads to that a change in a network function application requires only redeploying of the service which the change concerns, compared to the costs that redeploying of the entire application would incur. Another benefit is that different sets of services can customize a network to precisely meet customer demands. [4]

2.1.2 Virtualized Network Function

Network Functions (NF) of a mobile network are traditionally specified by individual functional blocks which are autonomous hardware components dedicated to a certain role [15, pp. 16-17]. This paradigm has become outdated in relation to current needs for flexible network adaptation and separated software and hardware development [15, p. 10]. Virtualized network function (VNF) has emerged as an alternative way to specify functionality of a network. The NFV group of ETSI has specified an architectural design of the infrastructure underlying VNFs, called NFV infrastructure (NFVI), and how VNF entities are generated on top of these infrastructure components [15, pp. 17-18]. The design classifies NFVI components into hypervisor, compute and infrastructure network domains according to their roles [16, p. 11]. Respectively, the virtual instance components are classified into management and orchestration and applications domains [16, p. 11].

On the context of this work, a basic hardware component of NFVI relates to a compute node. A compute node is essentially a unit capable of running a generic computational instructions set (host function) which can be configured to provide any desired VNF [15, pp. 18, 47]. The nodes are Commercial-Off-The-Shelf (COTS) servers, meaning servers that are built of standardized IT components and sold in high volumes [15, pp. 45]. The existence of a VNF depends solely on the host function, which is depicted as "NFVI container interface" between hypervisor domain and applications domain in Figure 1 [15, pp. 18, 47]. An example of a VNF could be a Mobility Management Entity (MME), which is a virtualized equivalent of MME core network function in LTE architecture [15, p. 31]. Figure 1 shows the architecture's overall division into domains and interfaces linking them. When a host function is configured to be a hypervisor, the compute node platform is able to host several virtual machines (VM), each of which capable of providing a complete or

constituent virtual function [15, p. 18]. Such one-to-one mapped VM providing a single constituent virtual function is called a VNF component (VNFC) [15, p. 10].

When a VNF consists of multiple VNFCs, their communication must be established with virtual links, which are required to give similar high bandwidth as obtained with dedicated hardware [3, p. 241]. NFV standardized networks should also scale according to the varying subscriber amount, meaning that deploying and removing of VNFCs and entire VNFs should be possible [3, p. 242]. The networked VNFCs can be placed at distinct compute nodes in which case the VNF is a distributed system [15, pp. 29-30]. In the architecture considered in this work, a single VNFC consists of a guest operating system (OS), on top of which a radio network application is run. The application implements part of business logic of a distributed VNF.

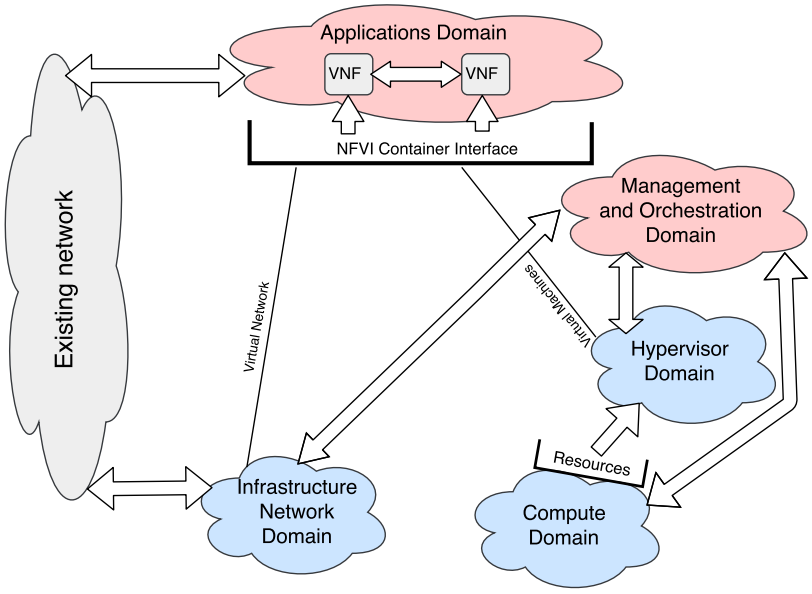


Figure 1: NFV domains and their relationships [15, pp. 27]

2.1.3 System interconnect architectures

Contemporary processor architectures embed memory controller onto the processor chip and divide the physical memory address space among the processors of a multiprocessor system. Each processor has a memory controller that manages accesses to a portion of addresses assigned to this processor [17]. A processor access addresses of its assigned portion via direct channel, yielding a high memory bandwidth i.e. the bandwidth between a processor and memory. Instead, accessing addresses of remote processors includes passing through a cross-chip interconnect, which has a lower bandwidth than the direct, on-chip memory-controller channel. Thus, processor memory accesses in this architecture are non-uniform, and non-uniform memory access (NUMA) system refer to a system following this architecture. [17]

The hardware architecture based on the described kind of system interconnections is called processor-centric network (PCN) [5, p. 145]. In a telco cloud, the PCN often

spans multiple compute nodes and further incorporates an intermediate Internet Protocol (IP) network. The memory of such system is distributed over physical nodes but serves as a single shared address space among nodes, called shared memory. The dedicated endpoints of the inter-processor links restrict scaling of the memory bandwidth. The communication over network and processing in the memory controllers increase the latency of remote memory accesses [5, p. 146] [17]. A request to a shared memory address of a remote processor is routed via other processor node and so the memory controllers are subject to shared resource contention [17]. Because of integrated memory controllers, a certain vendor's architecture, such as Intel QuickPath Interconnect and HyperTransport [5, p. 145], binds the supported configurations to specified processor and memory technologies.

In PCN architecture, a processor records the values stored at the memory locations of the shared memory to its local cache. A common practice for caches is to set them equivalent after a processor commits a write: such *cache coherency* is usually ensured with a protocol that specifies coherence requests to be sent via the interconnect. [18] The cache coherence requests are sent via network to other processors, causing them to contend on cache resources similarly to memory controllers [5][17].

The anticipated development of non-volatile memory (NVM) and optical networking technologies will enable a novel interconnect architecture, memory-centric network (MCN) [6]. In this architecture, NVM will replace DRAM and conventional I/O -rate storage as component essentials. Optical networking will speed up remote memory accesses such that shared memory access latency becomes roughly uniform. [6] Unlike in PCN, the path of memory accesses go directly from processor to the address in each occassion, thus giving direct access to the entire NVM address space for all compute nodes [6][7]. System with these attributes is described as *shared something* model which is portrayed in Figure 2. As the figure shows, the model assumes compute nodes to retain a local DRAM memory and let the system harness their NVM resources to a shared memory pool [6].

The elimination of mediating compute nodes in memory accesses leads to that access latency is solely processor-memory channel caused [6]. Accessing the shared memory from applications on different compute nodes is presumably non-coherent, meaning that cache coherency is not enforced at distinct cores in a write completion. This leads to that handling of memory sharing has to be included in the software where the access happens [6][7]. A subtle cache coherency, like one which issues updates on the basis of processor groups and prioritizes data with certain access patterns, can still come into question [6]. *Gen-Z* Consortium have developed a system interconnect supporting MCN architecture [19]. Their recently published specification about the interconnect is discussed in Section 2.2.3.

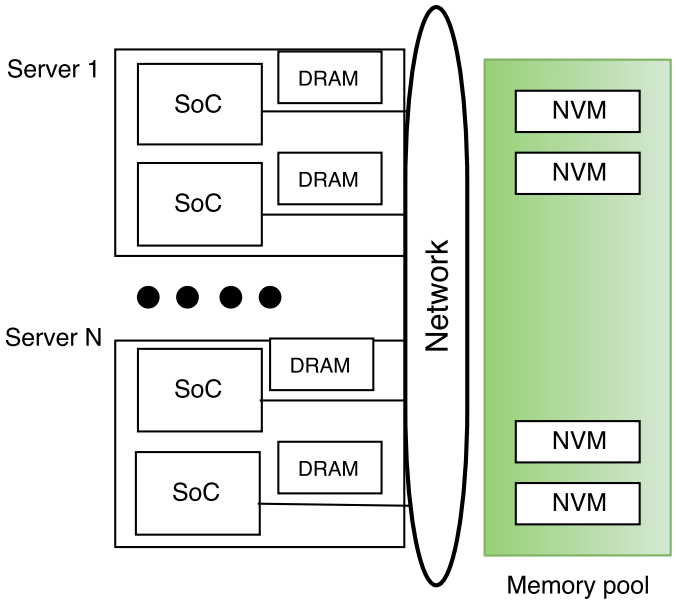


Figure 2: Shared something model [6]

2.1.4 Topologies and routing

Hardware components of cloud infrastructure need efficient communication to complete the access requests targeted to their resources. The bandwidth available for requests can be optimized with appropriate network topology and routing algorithm. Prevailing cloud data centers include different nodes: compute nodes taking care of computing, top-of-the-rack (ToR) switches offering communication between compute nodes and aggregator switches connecting the subnetworks established by ToR switches [20]. The network is typically arranged in a hierarchical topology formed of edge tier, aggregation tier and core tier [21]. Data centers used as part of a telco cloud are often organized in a similar topology, where edge tier spans compute nodes and ToR switches, and aggregation tier spans aggregator switches [21] [1, p. 456].

Compute nodes are referred as terminals in this section. VMs that populate compute nodes take regular two network hops to reach other VMs in the same rack while inter-rack communication traverses the source and destination ToR switches plus at least one aggregator switch. Current design has assumed that traffic is focused on the intra-rack site, and therefore the capacity of an aggregator switch has been fractional with respect to the capacity of a ToR switch. Thus, if a VM accesses resources of another VM, the latency depends highly on the distance between racks. [20] A standard ToR switch is equipped with 48 Gigabit Ethernet ports and up to four Gigabit Ethernet ports: the former is designated to intra-rack traffic and the latter to inter-rack traffic [21]. As previous switch design has obliged a tradeoff between the amount of supported bidirectional ports (radix) and port bandwidth, switches with a high radix have been considered as prospective design choice [22].

In the following text, a switch denotes a router or switch interchangeably. Fat-tree topology is a hierarchical topology which has been applied with high-radix switches

extensively [22]. The amount of tiers is indicated with L. A topology contains k pods expressing the count of ensembles of aggregation tier and edge tier, both of these tiers containing $k/2$ switches. k corresponds to the switch radix as well. The switching elements of the topology are identical at every level and so every link in the topology has equal bandwidth [21] Figure 3 depicts an example of a fat-tree topology with $k = 4, L = 2$ on its left-hand side. Fat-tree topology is rearrangeably non-blocking, that is, there exists a set of paths which deliver the full obtainable bandwidth for any kind of terminal node communication. [21] The maximum path length M of fat-tree topology equals $M = 2L - 1$ and the proportion of switches to terminals P equals $P = (2L - 1)/k$.

Ho Ahn et al. have developed HyperX topology which has many same favorable properties as fat-tree topology, but also some differences [22]. In HyperX, each switch is connected to a fixed number of terminals T and the remaining ports are for connections coming from other switches. A HyperX topology is specified with L dimensions and S switches beared by each dimension. Thus, every switch is given as a coordinate vector and connected to all the other switches that differ by one coordinate. [22] The bandwidth of a HyperX link can be a multiple of unit bandwidth which is commonly the terminal-to-switch bandwidth. With this flexibility, the bandwidth delivery between dimensions can be balanced. [22] A HyperX topology is characterized by a parameter tuple (L,S,K,T), where K denotes relative link bandwidth and S and K are generally in vector form [22]. The performance of a topology can be described by bisection bandwidth, meaning the total bandwidth from the links whose removal would result in splitting the network into two halves [23]. Figure 3 illustrates an exemplar HyperX topology with $L = 2, S = (2, 4), K = (2, 1), T = 3$ on its right-hand side.

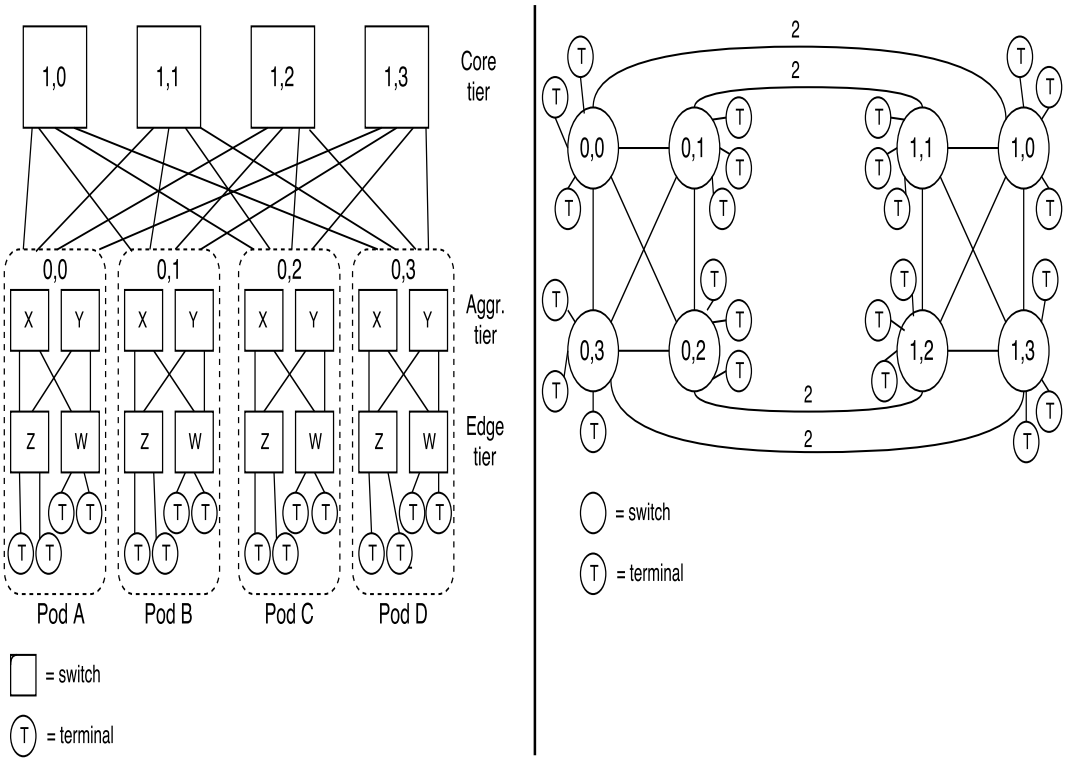


Figure 3: Example of fat-tree topology (left side) and HyperX topology (right side) [21][22]

HyperX was shown to perform competently with fat-tree topology when high-radix switches were employed and equivalent bisection bandwidth was maintained in the topologies. With switch radix 128 and 256, the results favored HyperX, while fat-tree was found to be more feasible with radix 32 and 64 employments. Fat-tree succeeds with low radices due to switches-to-terminals ratio: HyperX has to include a lot of lower-radix switches because number of terminals per dimension in it is more restricted than in fat-tree. [22]

Routing algorithm guides traffic around the network and possibly introduces overhead traffic, which affect on access performance. Often, a specific routing algorithm is developed for specific topology. For fat-tree, there is *Clos-AD* algorithm and for HyperX, *Dimensionally-Adaptive, Load-balanced* (DAL) algorithm [22]. Both of these are adaptive routing algorithms, that is, they consider network load among different paths when they determine where to route [22].

Clos-AD decides either minimal routing in decreasing dimension order or non-minimal routing at the source node of a packet [22]. DAL resolves a non-minimal route or if there exists none, resorts to a minimal route. The algorithm repeats this on each hop [22]. Minimal routing bases solely on shortest paths between source and destination, whereas non-minimal routing bases on uncongested, possibly longer paths between them [24].

DAL was measured to overperform Clos-AD both in average hop count and latency when these were simulated on HyperX topology with high network loads.

Although, the performance metrics of Clos-AD were determined with more incomplete information. [22] The equivalent inspection for congestion among dimensions showed DAL to perform better [22]. Another benefit of DAL is the capability to adapt to congestion while on the way, and because of this it can utilize the path diversity of HyperX more effectively than Clos-AD. Instead, HyperX topology has a disadvantage that the switch radix defines the boundary to the network size. [22] In the big picture, HyperX with DAL shows as a suitable configuration for high-radix switch network.

2.1.5 Quality characteristics of cloud storage

A cloud computing service is desired to have low latency and distribute effectively to access points while still being highly reliable in its operations [13, pp. 1,21]. Cloud applications run at different nodes in parallel to utilize the obtainable resources of the cluster [13, p. 22]. The above factors expect a cloud computing service to meet certain degree in the following quality-related characteristics:

- Consistency
- Availability
- Reliability

Telecommunication utilities are largely excepted to meet the same characteristics, which is elaborated in Section 3.4.2 [3, p. 241]. The definitions of these characteristics in this chapter apply to the rest of the thesis.

Conventional cloud systems have a PCN, detailed in Section 2.1.3, as their hardware back-end. This presents challenges in maintaining *consistency* on the shared memory accesses made by processes at separate distributed system nodes [13, pp. 44, 100]. Consistency signifies that a request to access shared memory should appear as a single operation in the system and each such request should be responded individually. More specific term to this guarantee is atomic consistency. [25]

Availability means that a node in operating state should terminate every arriving request procedure by returning a response to it. A strongly consistent distributed system cannot be highly available at the same time: for example, when a sequence *{write, read}* is executed and the message exchange related to *write* fails, the system cannot reflect the results of that *write* and the response is inconsistent regardless of possible successful termination of that *write*. [25] *Reliability* can be defined in cloud domain as the ability of the storage infrastructure, namely cluster nodes, to perform read and write requests directed to its resources [26]. If reliability of a distributed system is low, it often has availability or performance implications as well because the system should cope with fewer nodes to offer the service [13, pp. 84, 265].

Cloud providers with a conventional cloud often try to improve the discussed characteristics by applying replication in their distributed system [13, pp. 27, 265]. Replication is in its simplest form distributing a complete copy of an information object to other nodes of the cluster [13, pp. 27,70]. It makes service more resilient against primary storage failures but requires more work from the distributed system to preserve consistency [13, p .84].

Employing MCN, described in Section 2.1.3, as cloud back-end will change the characteristics of the distributed system overlaying the cloud. Memory management functions are predicted to shift from compute node OS into components at shared memory side. These components, namely memory controllers and accelerators, are indicated by "OS Services" in Figure 4 which depicts the mentioned shift. [6] Because of this shift, accesses to any segment of the shared memory will become equivalent from any compute node's point of view in this model and consistency likely turns to a rather minor concern [6]. Venkataraman et. al studied a storage effectively similar to MCN back-end: they stored data to a new type of structure which abolishes distinction between volatile and persistent copy of data. Each update on this structure generated a new version of it, overwriting of old versions was precluded during an update and the most recent consistent version was globally known. [27] The group showed by measurements that consistency was allowed with acceptable overhead from the versioning [27].

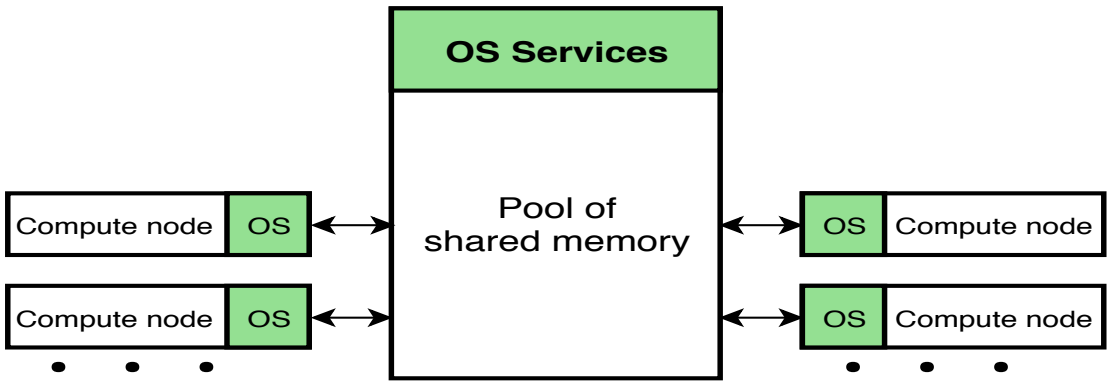


Figure 4: The shift of OS functionality in a cloud using memory-centric back-end [6]

Availability depends on the memory controllers' ability to know about the operating states of requested memory segments [6]. When a memory failure is detected in a segment to be accessed by application, the compute node OS should receive a signal about the failure and have a handling procedure to remain available in the context of the application. The signal would expose the parts of memory that have to be contained and its sending is a duty of collaborating memory controllers [6]. Reliability involves more complex issues than earlier systems as the memory device owned by a node can fail independently of the actual node [6]. The distributed system should have a means to maintain accessibility to data in failed memory, for instance by means of redundancy [6].

2.2 Technologies

Data access performance of the components in the new storage system architecture depends on the employed technologies. The accesses happen usually from a network application running on any computing unit of a cluster, which requires equivalent shared memory accessibility from all the units. Communication between applications

is frequent, making data transport between the computing units important as well. While operators desire access performance to be high, they wish costs to stay on acceptable levels. Next sections first explain essential computer system concepts in "Memory-related system concepts" section and then proceed to explain technologies associated with data accesses in the new architecture. "Non-volatile memory" is a technology that addresses the memory accesses, "Gen-Z" is a technology that enables memory-centric system level communication and "Optical transmission" discusses transmission technologies. Additionally, a section about research on another memory-centric technology, Hybrid Memory Cube, is included.

2.2.1 Memory-related system concepts

OSs commonly divides the virtual memory address space of a process into pieces of fixed size, which are in this context called *pages*. Similarly, the physical memory of the system is seen as fixed-sized slots by the OS, termed *page frames*. To combine these two for managing address spaces, the OS typically maintains a page table for each process. The page table records virtual page to physical page frame mappings for each page in the process' use. [28] In Linux OS, a process maps page frames at a file or device to its pages by calling *mmap* system call, and accordingly creates a new page table mapping [28][29]. Subsequently, the process can access the physical memory via pointer returned by the mmap call [30, p. 3]. The page-based management is generally called paging [28].

The OS might relocate pages to a disk, namely, *swap* parts of process' address space. This allows pages to be freed up from RAM and to go beyond the address range of the system in virtual address space of a process. [28] A particular file containing relocated pages from RAM is called a *swap file*. It is stored at the disk file system. When a processor requests data from a memory address that is missed in its cache, the memory is loaded to the cache in units of cache lines [31, pp. 378-379]. When an OS needs to clear a processor cache, it makes a *flush* operation on the cache lines of the cache [32][33]. Flush operations are related to non-volatile memory in particular, as this technology bases on both persistent and rapidly accessible data [33].

2.2.2 Non-volatile memory

The current storage models of cloud services typically store presently needed data to RAM and longer-term data to hard disk [13, pp. 264-267][34]. This principle of treating persistent storage access to be slow has prevailed for long in system design, defining both hardware- and software-level specifications and practices. Physical interfaces from disk to a CPU are undoubtedly slower than CPU-memory interconnects and the overhead posed by the I/O stack of OS slows down the accessing as well. [35]

To eliminate this legacy and enhance memory access performance, the development of various non-volatile memory (NVM) technologies has proceeded [36][35]. NVM is a storage technology which persists the data, yet enabling access latencies proportional to CPU-DRAM rates, rather than CPU-to-I/O device rates [6]. Each byte in a

NVM device has an independent memory address, that is, NVM is byte-addressable memory [31, p. 360]. Unlike current block-oriented access, byte-addressability enables direct load/store path to the memory [6]. Some common NVM technologies include phase-change memory (PCM), spin-torque transfer memory (STTM) and memristor [36]. PCM storing is based on crystalline states of a chalcogenide metal layer [37]. STTM is based on the magnetic orientation of a layer in a magnetoactive component [38]. In general, PCM has turned out the most promising NVM candidate in studies [35][36].

NVM has been researched as an extension for DRAM as well as substituent for the entire main memory [36][33]. Caulfield et. al's study presented that PCM and STTM exhibit DRAM-comparable nanosecond access latencies when the excess from buses and memory controllers had been cut out. Furthermore, the research group experimented NVM for virtual memory use case: the effective memory capacity on an application's use was enlarged by NVM. This showed that paging parts of large memory working sets to NVM results in almost similar performance as paging correspondingly to DRAM. The study also adduced that commodity OS, such as Linux, add notable overhead to access latency, especially due to their file system component [36].

Zhang and Swanson assessed the performance of running storage-intensive applications with different storage types used as the underlying system [33]. They concluded that main memory NVM outperformed I/O rate storage (SSD, HDD) clearly and reached almost DRAM rates. The improvement was highest in scenarios where the application wrote or instructed a CPU cache flushing to the storage frequently [33]. The major factor on the improvement was a special file system used by the NVM main memory. It made the physical pages of NVM dwell in the kernel address space permanently, which implied that the stage of copying from the storage to the OS buffer cache was removed from the access operation. [33]

In order to benefit from persistence, the data seen as persistent in the application must be determinately stored in NVM. This requires flushing of processor caches to NVM, which occupies processor channels and hence, introduces a cost in access latency. The cost can be mitigated with appropriate flush operation reduction techniques, such as Selective Persistence Flushing proposed by Zhang and Swanson. This technique yielded vast performance improvements in their studies, especially for applications having strict consistency and durability requirements, such as database applications. [33].

2.2.3 Gen-Z

Gen-Z is a hardware-agnostic system interconnect which provides a common linkage between various computer components, including processors, memory, FPGA and storage [39, p. 30]. The interconnect centers on load/store operation and memory-semantic communication concepts [39, pp. 28-30]. The former means operations that point an application directly to the memory of the other end of the communication, and the latter signifies the transmission of messages related to load/store operations made via the interconnect [39, pp. 30, 162-191].

The Gen-Z protocol specifies memory requests as communication means for Gen-Z connected components, referred as memory-semantic components. A memory-semantic request is understood as an operation on a component's resources which hides the explicit message exchange of the underlying microarchitecture [39, p. 32]. The operation is for example read or write. The requests are transported in packets and exchanged asynchronously between components [39, p. 28]. A memory-semantic component contains the logic to interpret a request based on its context, which leads to that media-specific logic can be excluded from memory controllers of processors [39, pp. 35-36]. References to a component mean a Gen-Z connected, memory-semantic component in the rest of this section.

Components might act as Requesters, Responders or Request-Responders in various Gen-Z topologies. A Requester generates request packets, receives and executes the instructions enclosed in the corresponding response packets. A Responder receives and executes instructions from requests, and reciprocally generates the corresponding response packets. [39, p. 32-33] In a trivial example topology, a processor-integrated memory controller sends packets containing read and write requests on a memory component which does the requested operations and responds to the controller. The memory controller and memory component are connected with a point-to-point link. [39, p. 36] More complicated and large-scale topologies can incorporate packet forwarding nodes, such as switches, together with memory-semantic components [39, pp. 36-38]. The ensemble of all interconnecting links is named as fabric. Any number of physical layer technologies can be supported by each Gen-Z network link and a component can have multiple such links, each allowed to follow same or different physical layer technology [39, p. 31].

To support messaging about special operations, the protocol associates an operation class (OpClass) attribute to every request and response packet. Components might assign implicit or explicit OpClass to a packet: the former defines that packet might be exchanged only via separately specified interfaces while the latter allows exchanging via any interface of a component. [39, p. 128] The format of a request or response packet conforms to the OpClass of the originating interface, or to an explicit class identified by a special field in the packet format [39, pp. 146-147]. The other important fields of the packet format are payload field, which contains the possible response data, and address field, which identifies the target address of the memory operation in question [39, pp. 148-149]. An important field of explicit OpClass packet is the Congestion Field which is updated by a switch or a Responder in case it detects congestion [39, p .695].

The Gen-Z protocol handles resource access control and memory management. A user-space process knows about its accessibility on a remote resource via capabilities which are permissions related to a memory address under access control. A capabilities instance evokes a protected object from which a system derives the accessibility of an address during the process' existence. In this manner, the same address space can be shared between processes running at different nodes. A permission to use a resource is derived from capabilities information and address field included in a packet: for example, a Requester has access to a resource of Responder if a request packet sent by it includes a capability tag with true value and the address associates

to a capability-controlled resource. [39, pp. 102-103]

Shared memory of a Requester-Responder pair is handled with the help of memory management units (MMU). A Requester MMU maps memory pages pointing to the shared memory at a Responder into the Requester's local memory pages. After memory allocation, a user-space application at the Requester accesses the shared memory transparently via pure read-write -like instructions. [39, pp. 105-106] The concerned units and their associations are illustrated in Figure 5. A Requester can map resources from one or more Responders to its memory address space. A Responder could also employ an MMU to do translations from the requested memory addresses to Responder-local memory addresses pointing to the actual resources. [39, pp. 105-107] If a contiguous memory page at the Requester is tied to a single Responder, the page is said to be non-interleaved, whereas a memory page with multiple Responder associations is an interleaved page [39, p .114].

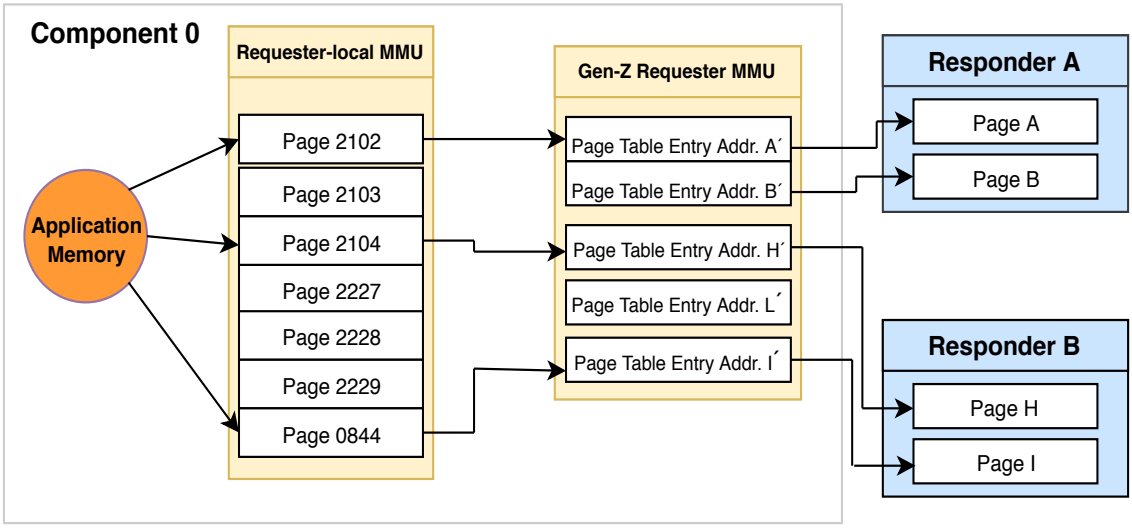


Figure 5: Example of memory access procedure between Gen-Z components [39, p. 105]

Gen-Z advances high performance through the overall fabric of connected nodes. Packets exchange is based on pipelining with asynchronous completion [39, pp. 69-70], which essentially means that each packet has only local knowledge about moving between stages of the transfer and completes independently of others [40]. Consequently, Gen-Z defines non-blocking communication that allows packets to move between the communicating ends with very dynamic rates and to have variable response latencies [39, pp. 69-70][40]. The aggregate bandwidth of a configuration can be grown by choosing the least occupied one among multiple paths which frequently occur with more complex topologies. A Requester-Responder pair can indicate a path interconnecting them as faulty via specific messages, which could improve resiliency of the topology [39, pp. 85-86].

For interleaved memory pages, the protocol uses a scheme which distributes the address mappings of a page in a balanced way across participating Responders. The

participants are configured in an interleave group which can consist of arbitrary amount of Responders. As access bandwidth becomes equivalent for all Responders, interleaved memory communication achieves higher aggregate bandwidth and lower latency compared to non-interleaved memory communication. [39, p.114]

Congestion can be tackled in switch-based topologies with components supporting explicit OpClass packets [39, pp. 49, 797]. The congestion source is detected and packet injection rates of such packets are decreased [39, p. 797]. An explicit OpClass -supporting Requester may be configured to have a Congestion Management Structure to adjust the packet injection rates [39, p. 628]. Such adjustment can be triggered by multiple retransmissions of a request packet without receipts, or with a Congestion Field value indicating congestion or distinct exceeding of expected packet exchange time at the Requester [39, p. 802].

2.2.4 Optical transmission

The increasing data utilization in the novel mobile network use cases require vast transmission capacity from the links connecting the compute nodes of VNFs [3]. It is well-known that higher the capacity, higher the transmission power and the energy consumed. Operators have an interest to minimize energy and equipment costs incurring from transmission [2]. Currently, optics is the industry standard technology to address capacity, energy and cost requirements given to the network equipment [41].

The cost minimizing and scaling of optical links has still been challenging in contemporary data centers, particularly due to limited utilization of wavelengths by optical interconnects [41]. The main optical interconnect of today modulates laser light directly according to bit stream values onto a single carrier wavelength [41][42, pp. 27-28]. This interconnect has yielded 25 Gigabits per second (Gbps) transfer rates per fiber [41].

An optical module with more advanced transmission capabilities has been presented in the research. This module consists of four such interconnects that are used in the current systems. [41] Each interconnect emits a different wavelength of modulated laser light, following an independent bit stream [41][42, pp. 26-27]. The resulting laser signals are multiplexed, which essentially means transmission of multiple signals on the same channel, onto a single fiber [41]. Hence, the module uses wavelength division multiplexing (WDM) scheme [42, pp. 26-27]. The scheme is specifically called coarse WDM, based on its relatively sparse 25 nm frequency allocation [42, p. 7]. A single fiber of the module can carry 100 Gbps rate, resulting from the summed rate of four current interconnects [41].

The described module would alleviate the equipment cost challenges as multiple optical links could be created with a sole fiber. The fabrication of the optical module is not overly complicated, which further advocates the cost-effectiveness. The energy and scalability challenges are also expected to reduce as switches could send signals to one port instead of current four ports with the new module. [41] HyperX topology design, elaborated in Section 2.1.4, has assumed intra-rack communication to be optical and hence supports the introduction of the new module [22]. It is thus a

suitable topology option to tackle the cost and scalability challenges of transmission in the future networks.

2.2.5 Research on memory-centric system interconnect

Hybrid Memory Cubes (HMC) have been introduced recently as an advanced DRAM technology for high performance systems. An HMC constitutes of vaults which, in turn, comprise a memory layer and a logic layer. The former layer embodies stacks of several DRAM resource blocks and the latter layer embodies logic to interconnect these stacks with other system components. One HMC builds on multiple vaults. An important part of the logic layer is a switch that interconnects the vault to vaults of other HMCs and processors. Messaging between CPUs and DRAM is done in a memory controller at the logic layer and messages are framed by a packet-based protocol. A message is a request, which the memory controller understands as a memory operation on a specific memory location under its supervision, or a response, which a CPU understands as data in return to a request [5]. Thus, the memory controller roughly corresponds to a Gen-Z Requester and a CPU to a Gen-Z Responder, both defined in Section 2.2.3, and the system interconnect between such components is an MCN interconnect [5].

An MCN configuration with HMC memory has been studied by Kim et al. [5]. In the study, the system components were connected in a distributor-based topology [5, p. 148]. Distributor-based topology determines that CPU nodes are directly connected to multiple HMC nodes and that each HMC node has connectivity to all the other such nodes [5, p. 147]. The switches in the logic layers take care of routing between the components, and a non-minimal routing algorithm was decided for the study. The specific routing algorithm used in the study was Universal Globally Adaptive Routing (UGAL) [5] [43]. As the CPU-to-memory communication in this setup has multiple path starting and finishing options and non-minimal routing was employed, path diversity was fully utilized in the experiment [5]. The performance measurements showed that CPU-to-memory traffic reaches substantially higher access bandwidth in the described MCN setup than in a PCN with minimal routing, and that latency on the path was lower [5, p. 153]. Instead, CPU-to-CPU traffic latency was significantly higher in the MCN setup than in the PCN setup [5, p. 153].

3 Radio network application domain

Making mobile networks cope with the incoming performance demands requires adaptable and flexible functioning from all parts of the network architecture. In the most recent 3GPP network deployment, Evolved Packet Core is responsible for traffic flowing between user equipment (UE) and packet data network (PDN), as well as administering of it [44, p. 25]. The corresponding part of the client's 5G mobile network is called cloud-native core network [4]. The design of the overall 5G network is radically different than earlier, with major reformations to apply Network Function Virtualization (NFV) and Software Defined Network (SDN) [4]. Cloud-native core network provides a backbone to these. It is logically formed of a customizable set of VNFs and a data repository common for all of them. The data layer segregates control plane and user plane traffic from each other. It allows a packet flow to have less network hops and gain higher bandwidth on its core network passage [4]. Both factors contribute to a lower end-to-end mobile network latency [1, p. 22]. The data repository must associate a storage back-end which is essential to be accessed with low latencies from VNFs.

3.1 Cloud-native core network

Cloud-native core network complies with the ETSI NFV architectural framework discussed in section 2.1.2. It also aligns with 3rd Generation Partnership Project's (3GPP) 5G specifications [45]. Cloud-native core network contains all the components making possible to include any VNFs to a mobile network [4]. The required NFVI compute domain components cover cloud-located servers, which can be client's proprietary servers or other COTS servers, storage resources and networking equipment [16]. The hypervisor domain components are based on a built-in KVM hypervisor kernel module of Linux and QEMU software, which virtualizes the underlying hardware for VMs hosted by the hypervisor [46][47]. Using KVM module requires the host system to be a Linux OS. The management and orchestration domain components manage the lifecycle of VNFs and maintain the VM assembly for each VNF by software [46]. Coordination between the mentioned domain components is made possible by open-source software, which has been OpenStack in recent Nokia releases [46][48]. OpenStack provides interfaces to configure available resources, provision VNFs with the resources and manage faults in the different domain components [46].

Each sort of VM in a VNF can be scaled in/out which subtracts or increases the amount of VMs from cloud, respectively [49][8]. Changing the amount of VMs changes the received capacity accordingly. Actual radio network applications run on VNFCs which are singular VMs as described in Section 2.1.2. The constituent VMs of a VNF are devoted to either control or user plane functions [4][49]. This separation gains addressing to plane-specific needs, as scaling and optimal geographic placement of VMs can be done independently for both planes [49].

The software architecture of the design conforms to microservice anatomy [50]. The anatomy categorizes software components to microservices and infrastructure services. A microservice component performs a single duty exclusively whereas

infrastructure services provide auxiliary functionalities for microservice components. This enables modular building of network applications: a network function, such as MME, becomes part of a microservice-style application only if the application subscribes the microservice of this function via an exposing API [1, pp. 131-133]. The exposing API consults Discovery Services, which discovers the location of running MME instance and associates it to the application. Another important infrastructure services are Data Services, which allows to share data among microservices and to persist their states [50]. The described discovery and data storage functionalities are defined to be part of 5G Core Network in 3GPP Release 15 [45, pp. 24,154]. The client has implemented Data Services as a special solution, named Shared Data Layer (SDL) [8].

3.2 Shared Data Layer

Analytics applications of a mobile network, such as base station software, measure and prepare various data for the purposes of VNFs, and also VNFs yield data that will be used by other VNFs of the network. Such data include subscription, policy, charging and session data. [8] When the low-latency use cases of 5G start to come true, the data set sizes to be analyzed will also exceed the capacity of conventional analysis tools [1, p. 337][2]. Such data sets are often termed as big data [1, p. 337]. In a conceived future network, an operator can open segments of its network to third-party services outside of its own business areas [2].

Currently, each VNF application (VNFC) acquires all the data it needs by itself and has its own storage to access the data. The data is possibly fetched from many sources, which causes a lot of signalling traffic to the network. In Cloud-Native Core Network architecture, Shared Data Layer (SDL) replaces these VNFC-wise means of storing and accessing data. It serves as a common layer for VNFs to access all types of data. [8]

SDL can provide all the extracted network and context information in a single version to VNFs [2]. In practice, SDL is a shared data repository maintaining different data types in their own classes [4]. Figure 6 depicts the SDL framework. The repository lives in a back-end storage which is accessible by multiple vendors, allowing operators to use VNFs from different vendors in their network. The common repository helps in managing the overall data set [8]. SDL framework offers open APIs for both storing and sharing data among VNFs, analytics applications and the possible third-party applications of the network. This enables that fetching and managing of network capability data can be elided from the logics of VNFCs, and local storages are not needed in the VMs of each VNFC anymore. Furthermore, VNFs can become stateless, since VNF session data comprises information about the state of VNF. VNFs access data from SDL via industry-standard protocols. [8] The analytics and third-party applications access SDL through northbound interface which hides the physical details of the back-end configuration from them [8][51, p. 5].

As VNF states are stored in SDL, a failure or difficulty confronted by a VNF can be dealt by activating another VNF which would retrieve the state of the struggling

VNF from SDL. Moreover, SDL induces that VNF resources are not linked as in traditional distributed system and thus a node failure recovery becomes lighter operation for the system. The renewed storage layer will offer very comprehensive data availability and low latency data delivery for VNF products. Resiliency of the overall core network also improves because of much simpler internal storing logic. [8]

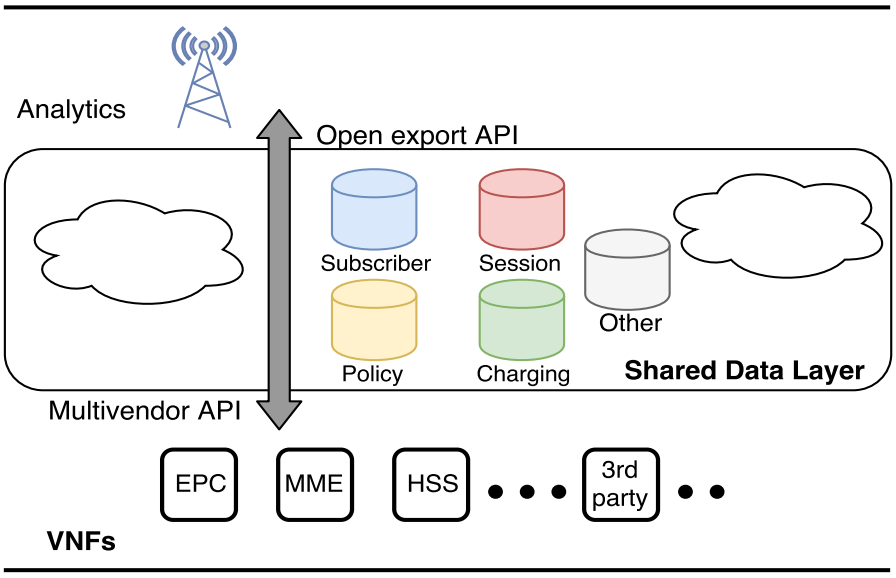


Figure 6: Shared Data Layer outline [8]

In tentative design, the SDL back-end storage has been plant to be based on distributed cloud computing infrastructure. Such infrastructure would ensure that the delivery of the network and context information is delivered to VNFs reliably and in a short response time upon its requesting. [2][4] In futuristic deployments, SDL might have multiple underlying back-ends and the applications would be able to choose the technology for their data accesses [49]. One microservice i.e. a radio network application would then run at a VM node and determine the back-end technology to take care of the physical storing for each data access.

3.3 Storage back-ends

Shared Data Layer has to be backed by a storage hardware to offer described data storing and sharing services in cloud-native core network. The storage can be practically utilized after a database management system (DBMS) has been configured on top of the hardware. In the next two sections, storage back-ends with different system interconnect architectures are described. In the third section, DBMSs are discussed with examples from both architectures. Redis is reviewed as a DBMS for PCN architecture and FOEDUS is reviewed as a DBMS for MCN architecture.

3.3.1 Processor-centric back-end

A storage back-end of present cloud platforms is based on the earlier described PCN architecture [13, pp. 241,264]. The client has employed a PCN-based back-end in its recent architecture releases. The back-end system can be run on servers that are physically distinct from VNF nodes, when they serve solely as a database for the nodes [4]. One database node might also be integrated to a server that already runs a VNF node. In the latter case, the storage and computing resources are more restricted in terms of both nodes, but the physical communication link between the servers can be excluded.

The design of the client follows the ETSI-specified ways of implementing storage infrastructure for the storage services. These ways determine two different setups: a setup where the compute and storage node are the same physical node running both client and server instances, and a setup where the storage node is a remotely accessed distinct node. [16, p. 43] In the former case, applications could access data from a co-located database exclusively, while in the latter case, the memory of the storage node would be shared memory accessed via IP network. The shared memory in the latter case would be typically accessed by multiple nodes, as VNFs commonly constitute multiple VNFCs which are distributed over many nodes. The hardware for storage nodes is either COTS servers or proprietary servers of the client [4].

3.3.2 Memory-centric back-end

Hewlett Packard Enterprise (HPE) has innovated *The Machine*, a platform of compute nodes attached to a memory which can be accessed by all these nodes. This fabric-attached memory (FAM) is a unified pool of memory, composed of NVM resources of the attached nodes [6]. Hence, The Machine is based on memory-centric network (MCN) architecture. Because of the yet immature NVM technologies, HPE has constructed the memory pool in its current The Machine prototype from DRAM resources shared by each compute node [7].

The compute nodes are supplied with an integrated circuit component, called a bridge, which manages routing of memory accesses among fabric-attached nodes. [7] Operation of this bridge component is illustrated in Figure 7. The stored data is persistent in FAM, and can be accessed with near DRAM-rate latencies. Because the memory pool is accessed non-coherently, data sharing and memory failures are to be managed in software level [7][6]. To address these issues, the developers have implemented a Linux OS with customizations on memory management, although, it is not in a very mature phase yet [7][10]. Furthermore, coherency matters has to be considered in user-space applications [7]. When a system of nodes uses FAM for the same purpose as a computer uses RAM, it is often referred as non-volatile random access memory (NVRAM). Computing with The Machine like platform is referred as memory-driven computing.

The system interconnect of The Machine should enable heterogeneous computing resources while sustaining DRAM-rate high performance [6]. Gen-Z, which was detailed in section 2.2.3, is an interconnect engineered to fulfill these properties [7]. The load/store operations specified by Gen-Z allow to bypass local processor

caches in remote node memory accesses [9]. The internal network of the back-end is desired to support low and uniform hop count for the memory accesses among attached nodes, which advocates to use high-radix switches with a suitable network topology. One such topology could be HyperX, described in section 2.1.4. [6] Hewlett Packard Laboratories has researched an optical module which has promised proper cost-efficiency and bandwidth for memory-centric communication [41]. This kind of module is assumingly integrated to each compute node in the matured The Machine platform. The modules will connect the bridge components of the nodes, making an optical network between them. [7] The connections to nodes outside of the home rack are still assumed to be based on Ethernet [7][9]. Although, an Ethernet interface of The Machine node can well be connected to the processor-memory part of the node via Gen-Z [7].

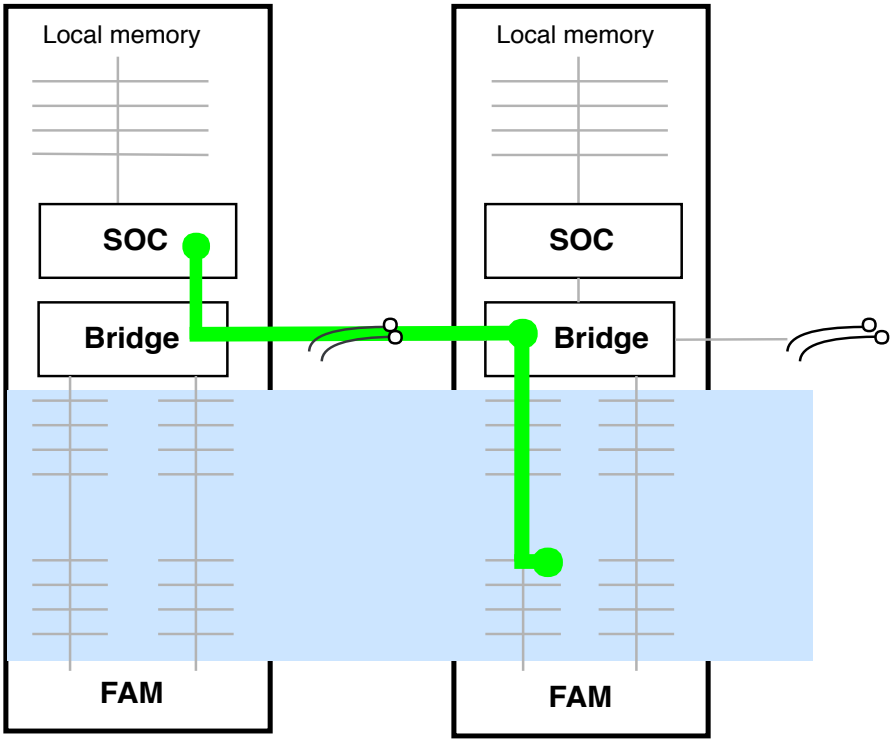


Figure 7: The routing of memory accesses among The Machine nodes [7]

3.3.3 Database Management Systems

Commonly, data is structured into records at the storage back-end, and the entire record set beared by a logical node is called a database. Databases are managed by a database management system (DBMS) software. In the discussed radio network application case, DBMS server is a component acting as an intermediary between SDL interface and underlying databases. The SDL interface is specified later in the "Experimental methodology" chapter. Data integrity, data accesses and I/O

concurrency control are handled by the DBMS [13, p. 245]. Thus, DBMS engages the databases in the back-end to VNF applications. Processor-centric back-ends have wide range of available DBMSs whereas memory-centric back-ends have not yet been widely targeted by database developers.

The processor-centric back-end is overlaid by a *Redis* DBMS in the current client architecture release [34]. Redis is open-source -based software that stores data in a key-value fashion, which is a popular schema in data stores of cloud use [34][13, pp. 264-265]. Redis is a NoSQL DBMS whose most important properties are that the database is normally stored in RAM and that consistency of concurrent database operations is not perfectly assured [34]. Redis is able to replicate and partition the database to other servers [34], allowing the shared memory to be located on one or several physical nodes. Redis serves connected clients; in the SDL use, VNF and analytics applications at VNFCs act as Redis clients. When such a client application requests data via SDL interface, a Redis server running node locates the data from its RAM or hard disk and transfers it to the application via TCP/IP communication [34].

To manage accesses to the radically transformed shared memory configuration of The Machine, a new kind of DBMS is a necessity. HPE is still studying the ultimate recommended system for this purpose. [7] One DBMS emerged from their research is Fast Optimistic Engine for Data Unification Services (FOEDUS) [7][52]. A FOEDUS server maintains an image of the database in the page frames of physical memory. The image is stored to both volatile DRAM and persistent NVRAM of the hosting compute node, such that both storages will contain equivalent page frames containing the database. Subsequently, a dual pointer pointing to both of these physical locations is generated as a handle to the database. [52, p. 693] The volatile page in DRAM is the latest version of the database and can be mutated, while the persistent (snapshot) page in NVRAM contains previous, immutable version of the database. Each compute node writes the committed operations to a log file whenever they manipulate the database in their local DRAM [52, p. 696]. On defined time intervals, the handler of the log produces a new snapshot page from the most recent snapshot page enforced with the manipulations accumulated in the log file. The handler does not generate a complete image of the database on these updates but only replaces the modified parts. [52, p. 696] Upon such snapshot update, the volatile pages in DRAM are replaced by the new snapshot page, reducing DRAM consumption. In this way, NVRAM will eventually consist of multiple timestamped versions of the database which are derived from each other, and DRAM holds the working version of the database whenever there are pending modifications [52, pp. 693, 696]. Figure 8 shows how the database image is stored and how the dual pointers are formed in a fabric-attached compute node. In the figure, *SP1*, *SP2* and *SP3* are different snapshot pages such that SP2 is constructed on the basis of SP1, and SP3 analogously on SP2. The records of FOEDUS database are in tuple format, which is essentially similar to key-value format used with traditional back-ends [52, p. 695]. In the SDL use, the FOEDUS server would be linked to VNF and analytics applications via SDL. Then, memory accesses done in the VNF application are able to be mapped to readings and writings of page frames at the node-wide NVRAM

storage and node-specific DRAM storages.

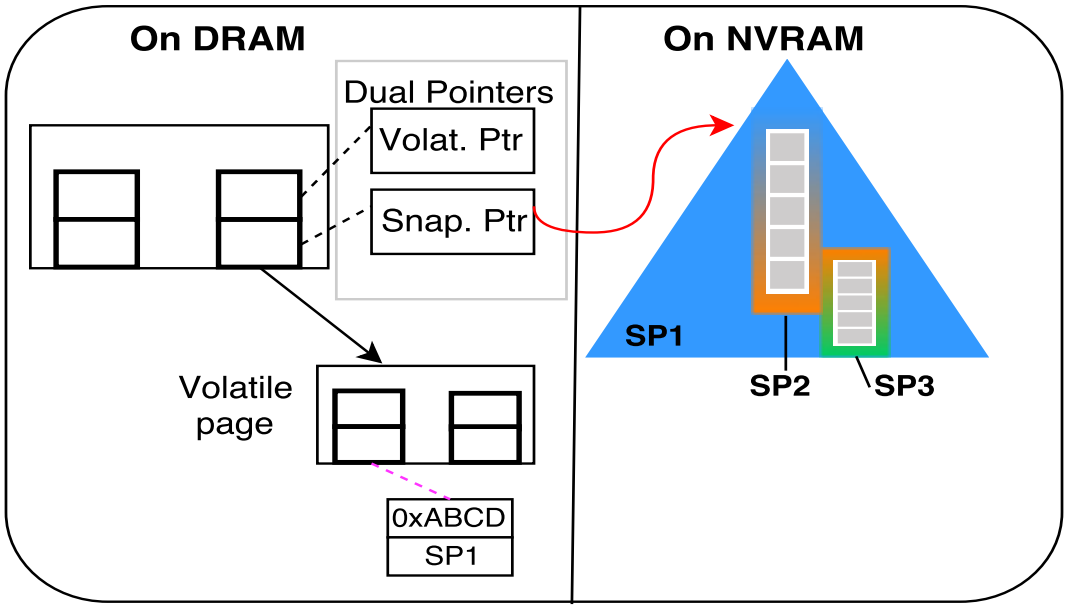


Figure 8: FOEDUS storage architecture outline [52]

3.4 Suitability of storage back-ends

Mobile network utilities, particularly user plane generated workloads, require high performance and optimizability from the underlying storage back-end [3, p. 243]. As VNFs are nowadays constructed modularly from microservices, many VNF components are likely accessing data from the back-end component simultaneously. The implementation of such concurrent VNFCs requires understanding about the consistency of the underlying storage in order to control execution and properly revive corrupted data objects [7]. The delivered service have also stringent reliability and availability norms: network outages should be unnoticeable and recoveries should be performed automatically. Their fulfilling is expected similarly from NFV-based architecture as from dedicated hardware [3, p. 239]. The hardware and software design of the storage back-end has impacts on fulfilling these requirements. This section discusses these impacts for a PCN back-end with Redis on top and for an MCN back-end with FOEDUS on top, both designs described above. The performance aspects are considered in the first subsection, and the consistency, reliability and availability aspects in the second subsection. Finally, an overview of the suitability of the back-ends is given.

3.4.1 Performance

The performance aspects are tightly associated with the client handling procedures of the DBMS, the compute node OS properties and the behavior of used network protocols. The DBMS software of the back-end should have a concurrency model to

define how the concurrent I/O events, i.e. reads and writes, are managed [53]. In the following text, client applications of DBMS servers refer to individual radio network applications running at VNFCs.

Redis server is a single-thread program, which controls concurrency by locking the requested data entity, implying that only one client application instance at a time is allowed to read or modify the entity [34]. Thus, Redis might cause a client thread to block, meaning a situation where a thread yields CPU to a different task than the thread in question [53]. The shared memory of PCN architecture makes developing of efficient programs difficult due to stalling coherence signals which easily raise when programs access the memory in parallel [9]. If FOEDUS server would be deployed on top of The Machine, every database operation would be executed by an individual thread and the program could be run on multiple threads [52, p. 695]. The concurrency model of FOEDUS allows client applications to read/write concurrently as the applications maintain run-time modifications in the DRAM of their local nodes, while the most recent database image can be continually obtained from NVRAM by any node [52, p. 695]. Consequently, FOEDUS server enables non-blocking execution of threads. Non-blocking execution cuts latency of data delivery from the back-end to the VNF application.

Chen et. al showed that an early prototype of The Machine based on a multisocket computer overperformed a highly advanced, PCN-based graph processing system by orders of magnitude. The systems were tested for accessing Gigabyte-scale array data. The group used a database engine on top of the FAM that loosely corresponds to FOEDUS DBMS: the global shared states of array items were stored to FAM, allowing to omit a lot of expensive shared data update routines among the nodes. [10] Redis exchanges data requested by client applications via conventional IP communication. The data flow is controlled by TCP/IP stack, which is a built-in feature of kernel of the OS of a Redis node. The stack restricts the processing speed of requests due to its costly operations, such as system calls and copying of buffers between user-space and kernel space. [54] In MCN, the communication happens presumably with direct load/store instructions via Gen-Z interconnect, which would eliminate TCP/IP stack overhead exhibited by the OS [7][6]. However, the OS should still manage the I/O occurring in both Gen-Z and auxiliary IP network interface, without interfering with the communicating user space applications [6].

In a multi-node PCN back-end deployment, a node has to indicate about a dropped packet on the IP network by sending control messages in packets. Corresponding MCN deployment can omit control messages as memory-based transmission is lossless and thus traffic in the main transmission medium is reduced. Nevertheless, the data sent over FAM may still be lost because of failures in NVM hardware. [9][6]. Another source of overhead network traffic in PCN deployment is the cache coherency protocol which makes processors issue at least cache value invalidation and cache value update messages to the network [9][18]. Schlansker et. al found that The Machine architecture fits well for NFV purposes as it allows to access persistent data fast and reduces redundant copying of network packets [9]. Data contents can be passed as pointers within load/store packets, in contrast to sole copying of full packets between nodes in conventional shared memory. Hence, a single store instruction from

an application could reach multiple remote applications at The Machine nodes and a node can judge the need for a full copy of a packet from the received pointer. [9] The commonly accessible FAM makes MCN architecture better in terms of scalability of VNFs: the applications can access shared data with a single operation instead of earlier multiple copying and transporting of data [9].

3.4.2 Quality characteristics

Consistency of a back-end is affected by how the back-end ensures equal shared memory views among back-end nodes. This is called as consistency model. Reliability concerns the fault and workload tolerance of the back-end infrastructure components [13, pp. 84,100]. Availability is majorly affected by how a node can obtain data from shared memory in any system conditions. Faults in software or hardware have implications on consistency and availability as well.

As a NoSQL system, Redis follows eventual consistency model, which works such that a database record is guaranteed to be consistent at some time in the future instead of at the moment the record is updated [13, p. 265]. In contrast, non-coherent memory of The Machine does not guarantee consistency when shared memory operations are committed at distinct nodes. Particularly, a store operation targeted to a remote node is not guaranteed to be visible for a load operation to the same memory location at that node. Such an operation is guaranteed to be consistent only if the pair of accessing applications would be running at the same node. [9] Therefore, the functionality corresponding to cache coherency protocol has to be explicitly programmed in VNF software [9]. However, volatile caches and persistent NVM make this difficult, as node or software crashes would retain the corrupted values at NVM while cache values would fade away. Recovering from such occasions requires insight on data flows among caches and shared memory [7].

In The Machine, programming of persistent data might be ultimately implemented with runtime environments allowing applications to access memory in lower level than today [7]. For the time being, consistency model of The Machine does not guarantee similar consistency as the model of Redis. Although, there are no implications that consistency would be critical for VNFs in many use cases; one exception is storing of individual VNF instance configurations into SDL [49].

Managing shared memory in the program level often proves to be laborious because of the situations requiring locking of the shared data. Any failure leading to a crash of a thread that has locked a shared object would result in permanent locking of the shared memory resource. This causes the system to be unavailable for operations on this resource. A way to avoid permanent locks in threads and processes is to use messaging. Messaging would facilitate continuation of other program executables in case of an exception, such as a failure, in a single program module. With plain IP communication, there exists fairly established messaging mechanisms, whereas for memory fabric communication, there does not exist well-known mechanisms yet. At least Hewlett Packard Laboratories has ongoing development of services and APIs to enable messaging via FAM. [9]

The NFV specifications state that COTS hardware from any vendor can be used

and combined, which might cause hardware faults and further, VNF malfunctioning. Other NFV-specific fault sources include software and operator misconfigurations. Failures in the compute node network stay the same from the earlier design [55].

In PCN network, all data accesses to shared memory are managed by OSs at compute and storage nodes, and handling of faulty communication majorly conforms to well-known procedures, such as TCP retransmissions [6][54]. On the other hand, some resources at the storage nodes are probably highly requested by VNFCs, and thus a certain node and the network path to it might run into overload issues. Such overloading might cause discarding of requests to these resources and more severely, software crashes at the storage node. With MCN back-end, the VNFCs access data from other compute nodes via memory-centric system interconnect and the accessing is likely to be frequent. Memory is anticipated to be managed by auxiliary elements, such as memory accelerators and controllers [6]. These elements will be similarly individual failure points as storage nodes in PCN, but without the responsibility of serving database requests from their own memory. Thus, their failing does not affect reliability as badly as failing storage nodes in PCN [6]. The supposedly employed Gen-Z interconnect, described in Section 2.2.3, will enable very rapid data exchange and mixing of different hardware components. With this type of interconnect, the size of the memory attached to the fabric presumably grows and shared memory errors become more frequent. A load/store access to a location at faulty memory is probably not desired to cause halting or interrupting of VNF processes, thus, handling some errors as exceptions would be a considerable practice. Such overlooking of errors has to be balanced in the MCN back-end to achieve sufficient reliability. [6] As the memory in MCN is intended to be non-volatile, clearing human or software errors from the memory does not work out by simply rebooting the node [6].

3.4.3 Overview

As multiple cores are commonplace in the commodity server processors of today, the maximum computational power from CPUs is gained by using multiple threads in the server side of DBMS [1, p. 130][54]. Non-blocking execution of threads reduces the latency of delivering data from the back-end to the VNF application. The sizes of data blocks that VNFs access are relatively small. MCN with FOEDUS has capabilities to process such accesses efficiently while Redis with PCN spends most of the processing time on TCP/IP stack operations. The current OSs does not seem to be adaptable enough for the very high packet processing rates required by NFV applications [9]. However, special packet processing software, such as Data Plane Development Kit (DPDK) by The Linux Foundation, might heal data flowing between the applications and network components [56][54]. The fabric interconnect of MCN offers much more cost-effective data exchange between VNFs than PCN because of streamlined interconnect protocol.

Several programming models for dealing with the new memory configuration are already being developed [57][58], but a standardized way to make both volatile caches and NVM interwork with software has yet to come [6]. Consistency is not similarly supported with MCN back-end than with PCN back-end, and thus the application

programs running in an MCN-backed network cannot rely on that. This has to be considered in the design of certain applications. PCN back-end has single points of failure in its software and network paths, which are the main influence on its NFV reliability and availability. Instead, NFV reliability and availability in MCN back-end are mainly influenced by the reliability of NVM and how different errors on NVM are interpreted by the back-end. In order to achieve the required high availability and reliability, MCN provides more performance and scalability benefits than PCN but introduces a lot of unfamiliar methodology. The challenges raise from mapping the new methods to NFV domain, such as how radio network applications adapt to the shift from volatile data to persistent data. In the end, the storage service should be highly available and reliable from VNF point of view, regardless of what kind of methods have been used on the underlay [8].

4 Experimental methodology

The aim of the experiments is to simulate data accesses performed by a VNF component in a real mobile network environment. An application is a simulation of radio network software that could be running on a VNFC and provide a part of network functionality in an existing mobile network. The applications shall comply with cloud-native core network architecture, and therefore, their data accesses should happen in a similar way as if Shared Data Layer (SDL) would be in place. The main point of the experiments is to compare the performance of such accesses between a processor-centric back-end deployment and a memory-centric back-end deployment. Because of fully-fledged memory-centric back-ends are yet to be available, the memory-centric back-end is simulated with a surrogate platform. Several means to generate a reasonably similar platform are available; they are discussed in "Back-end simulation methods" section [59][60][61]. The focus is to have similar characteristics with The Machine prototype back-end described in Section 3.3.2. The latter "Measurement methods" section discusses the software related to SDL and performance measurements.

4.1 Back-end simulation methods

Computes nodes in The Machine platform are planned to access shared memory via NVM resources. On the current prototype platforms, NVM hardware is still replaced with memory splitted from DRAM capacity of the compute nodes [10][52]. To access such shared memory pool from a VNF application via SDL, a DBMS that suits well on top of memory-driven computing platforms is a necessity. Such DBMSs were still under research and could not be obtained during this work, and thus any DBMS with shared memory support was searched for the experiments. The back-end to be used was desired to exploit the technologies and algorithms used in The Machine, which are discussed in sections 2.1.4, 2.2.3 and 2.2.4. A platform that follows a system communication protocol similar to Gen-Z was obtained from HPE for the measurements. This product is called *Superdome Flex* and the obtained platform is detailed in "Superdome Flex" section. However, a stand-alone Superdome was not enough for the experiments, and thus available simulation tools were studied also. The studied tools were *Fabric Attached Memory Emulator* (FAME) and *WhiteDB*, which simulate the accessible shared memory of The Machine, and *SuperSim*, which simulate the potential interconnection network topologies and complete VNF application activity [60][62][61]. The former simulator tools are described in "Memory-centric shared memory simulators" section, and SuperSim is described in "Memory-centric networking simulators" section. Finally, the complete surrogate platform decided for the measurements is reasoned and presented in "Decided back-end surrogate" section.

4.1.1 Superdome Flex

Superdome Flex is an in-memory computing platform developed by HPE. It is composed of one or more chassis units. One such chassis, called a Base Chassis,

contains four CPU sockets and 48 DDR4 RAM slots. A Base Chassis is part of every Superdome Flex system. [59] Each chassis is equipped with two Intel Xeon Scalable 81XX or 61XX processors which provide 28 cores to all four sockets. Each socket has 38.5 MB of last-level cache capacity, which is shared among the 28 cores of a socket. [59][63] In a multi-chassis configuration, chassis are interconnected via cabled crossbar fabric, called HPE Superdome Flex Grid. The cable material is copper. The hardware component that enables communication to chassis external to a Base Chassis is called HPE Superdome Flex ASIC. Every chassis has two such chipsets and each CPU socket within a chassis is connected to either of them. The chipset is linked into similar chipsets in other chassis via Superdome Flex Grid cables. The linked ASICs use adaptive routing to route the traffic to the fastest available path and to balance workload over the fabric. [59]

The Superdome product has been used to study memory-driven computing yet because of The Machine prototype platforms are still on their development phase [7][10]. The important differences between the platforms are cache coherency and the system interconnect protocol in use. In an ideal The Machine prototype, memory is accessed non-coherently across the fabric, while in Superdome Flex, Flex ASICs maintain cache coherency over all processor sockets. Superdome Flex also uses a system interconnect protocol which is not a complete equivalent of the memory-semantic Gen-Z protocol to be eventually used in The Machine platform. The hardware architecture and configuration provide three different levels of memory access latency between sockets: these are local, direct and indirect attach. [59] Hence, the latency seen by the sockets is not uniform and they can be called NUMA nodes.

For the experiments of this work, a remote access to a Superdome Flex platform with 8 sockets and two chassis was obtained. The size of a single RAM module was 64 GBs, making the size of a single NUMA node RAM memory to be 768 GBs. The platform was running a single Debian-based Linux OS, and all the experiments were run on the kernel of this OS. The eight NUMA nodes of the platform are identified with numbers 0-7, with first four being in the Base Chassis and the last four in the external chassis.

4.1.2 Memory-centric shared memory simulators

HPE provides an open-source developer tool FAME which imitates the memory pool with Linux shared memory (SHM) and offers a roughly similar programming environment to the environment of The Machine -attached node [60][64]. Another open-source project, called WhiteDB, offers equivalent SHM memory with a NoSQL DBMS on top of the shared memory, enabling to access data by straightforward API calls [65]. FAME nodes are QEMU VMs, which are managed by KVM hypervisor on the host system, and see the SHM in their virtual physical memory [64][47]. WhiteDB nodes are similarly bind to exist within a single host system, but both host and node OSs can be chosen more freely than in FAME as they are not bound by Debian dependencies of FAME [60][62].

The SHM accessing in FAME is based on a QEMU feature called Inter-Virtual Machine Shared Memory. This feature exposes a region of physical memory of the

host system to the VMs via pseudo Peripheral Component Interconnect (PCI) device. The PCI device points to allocated page frames, in practice, to a regular file, at the hard disk of host system. [64] In WhiteDB, nodes see shared memory similarly as in FAME, but each created database allocates a new SHM segment from the RAM of the host, instead of hard disk [30][62, p. Tutorial]. Thus, an SHM segment can be allocated in the application software, unlike in FAME where the file is one static SHM segment [62, p. Tutorial]. Every database is uniquely identified with a key in the scope of a single host system [62, p. Tutorial][11, p. 923]. Such key is called an SHM key. The tools use different shared memory API: FAME uses *POSIX shared memory* API, while WhiteDB uses *System V shared memory* API [30]. The essential system call of the former API is *mmap* and of the latter API *shmget* [60][65].

To conclude, a VNF application at FAME-attached node would access a shared physical address space associated with the backing file at the host, whereas an application at WhiteDB-attached node would access a database on a specific SHM segment via WhiteDB API calls [64][62, p. C API]. WhiteDB uses database-level locks to control concurrent write accesses by processes: when a process has requested to do a write on the database, other attached processes are refused to access the database until the requester is done [62, p. Tutorial]. WhiteDB has also been studied with other popular NoSQL databases in terms of speed [62, p. Speed]. Because of the more practical methods to access data and broader earlier studies, WhiteDB was selected over FAME as the shared memory simulator candidate.

As memory segment keys are only known in the scope of a kernel, WhiteDB using applications require a run environment which is associated with the kernel [11, p. 923]. Such environment is either a container, or a native OS, which practically means that the application has to be a process that is spawned in the native OS. In this work, containers refer to *Docker containers*, which are nodes that enclose just a group of user-space processes on the host OS [66]. In both container and native process deployments, the host OS should be a Linux distribution which has all the programs needed to install WhiteDB software from source code with standard building tools, and to compile C and C++ source code. These programs include git, GNU autotools, and a standard C compiler, such as GNU C compiler [62, p. Install]. The commands to install all the packages required by WhiteDB and make the preparatory settings for using WhiteDB library in Debian-based OS are provided in Attachment A. In the container deployment, the Linux is additionally required to be supplied with sufficiently new version of Docker Engine [67].

The OS of the container and the packages to be installed are specified by instructions inside a Dockerfile. This file is read upon running a build command to build the container. [68]. When containers are used, the WhiteDB shared memory database must be created on the host machine before launching the containers for running the simulation program. The containers are configured to mount the location of the shared memory at the host file system, after which an application at a container can attach the database by invoking a WhiteDB database attach call with the same key as used in the database creation call at the host [62, p. C API].

4.1.3 Memory-centric networking simulators

Nic McDonald's *SuperSim* is a simulation framework that can simulate interconnection networks with a specific workload (application model) [61]. Interconnection network is simulated as a network model. The application model is run as part of the main simulator program and controlled through that, rather than as independent processes running on system instances. [61] Both network model and application model can be specified comprehensively. The most essential network model specifications are the network topology, protocols used and the technicalities of the network components. For application models, the most relevant specification is the packet injection rate. Any application model and network model can be mixed in a simulation. [61] SuperSim includes a tool that runs all the tasks a simulation consists of with all the hardware available in the simulating system [69, p. Basic Simulations]. The simulator has been used to explore routing within systems optimized for very high-radix network designs [61]. Thus, it could suit for exploring optimal network topologies and routing algorithms for a memory-centric back-end in the SDL use.

SuperSim can be run in a machine with a sufficiently new Linux OS that supports up-to-date versions of ordinary packages, such as Python3. To specify and run simulations, the simulator requires a few such packages to be installed, followed by the building of SuperSim project from source code. [69, p. Installation] Brief instructions for fulfilling these actions on a Debian-based Linux are provided in Attachment B.

4.1.4 Decided back-end surrogate

The memory-centric back-end surrogates were narrowed down to two: a Superdome platform with WhiteDB configured on top and an entire simulation of assumed interactions between application and back-end with SuperSim. An important decision aspect was to get a setup that produces desired measures quickly. The setup should also be able to produce the desired measures without extra components, and the ready-made SDL API and test applications are preferred to be capable of being integrated to the setup. The SDL API and test applications are presented thereinafter in "Measurement methods" section.

In WhiteDB back-end, compute nodes are either containers or processes that share the system resources of the host machine [47]. In a production back-end, a compute node would probably have more dedicated CPU and RAM resources. The radio network application could be still similarly based on a single process. An application that runs at a compute node of WhiteDB setup could access the mimicked FAM i.e. a region of RAM at the host system directly, meaning that inter-processor data transfers are not causing additional access latency. The latency caused by mediating OS is cancelled as well, since the kernel of the host system is shared for these nodes and the System V shared memory calls used in WhiteDB database accesses are not mediated by the kernel [66][30, p. 5]. Due to this, a VNF application running at a WhiteDB node can see the data records inserted by other nodes virtually immediately [11, p. 997]. The "route" of WhiteDB accesses to the memories of internal NUMA nodes goes from the CPU to the RAM via on-board interconnects, while routes to

external NUMA nodes include the copper between chassis in addition. Thus, the memory attachments are internal attachments among processes or container process groups of a system, instead of external attachments between physical devices [11, p. 922]. The single OS likely omits some overhead compared to passing messages between multiple OSs. As access latency in The Machine is only caused by optical cables and the bridge linking the memories, the latencies are caused by different components in the WhiteDB surrogate and in The Machine [7]. Although WhiteDB does not include a server process in the same sense as traditional databases, the lock-based shared memory resource makes its concurrency model close to that of Redis [62, p. C API]. This is different from those databases that have been used on top of other The Machine surrogate platforms used by HPE [52][10]. Despite the imperfections, an access from a WhiteDB back-end node can be seen to be largely similar to a Gen-Z load/store instruction made in a genuine The Machine.

SuperSim is run on a single system instance. The hardware resources of this single system do all the computing related to the simulation run. As distinct system instances are not required to run simulations, the simulator scales well for networks with high compute node counts. Also, since the simulator utilizes hardware optimally, running many simulations consecutively is likely to scale well. The SuperSim application model generates the nodes present in the simulated network and determines the traffic patterns between them. [69, Basic Simulations] SuperSim can model the accelerated network bandwidth of the real back-end, but the actual memory operations at nodes are not modelled. Furthermore, the applications can not be explicitly instructed to make a specific shared memory access but they are merely sending specified type of traffic around the network. Obviously, there are not any DBMS modelled either. The simulator can produce result data about network metrics seen during a simulation, which are associated with specific part of the network, such as a single link [69, Basic Simulations].

WhiteDB provides access to a real shared memory from individual applications while in SuperSim it is a matter of how well the software model fit to shared memory accesses in production environment. SuperSim does not either offer a ready Gen-Z -like protocol implementation but such implementation should be implemented by own. In VNF production environment, applications will be running on individual system instances and the SDL API accesses require data to be structured. As the experiments were desired to cover real world factors as much as possible, WhiteDB was decided as the simulator tool for the experiments. The nodes were decided to be native processes at the Superdome Flex because of the differences of technical details in attaching shared memory segment to a native process and a process at a container. A hardware setup with an example WhiteDB database attachment at the RAM of NUMA node 7 is illustrated in Figure 9. In the figure, "Host OS" indicates that a single OS spans both chassis, and the grey color indicates where the WhiteDB database is stored. The figure describes the main parts of the specific Superdome Flex platform used in the experiments of this work.

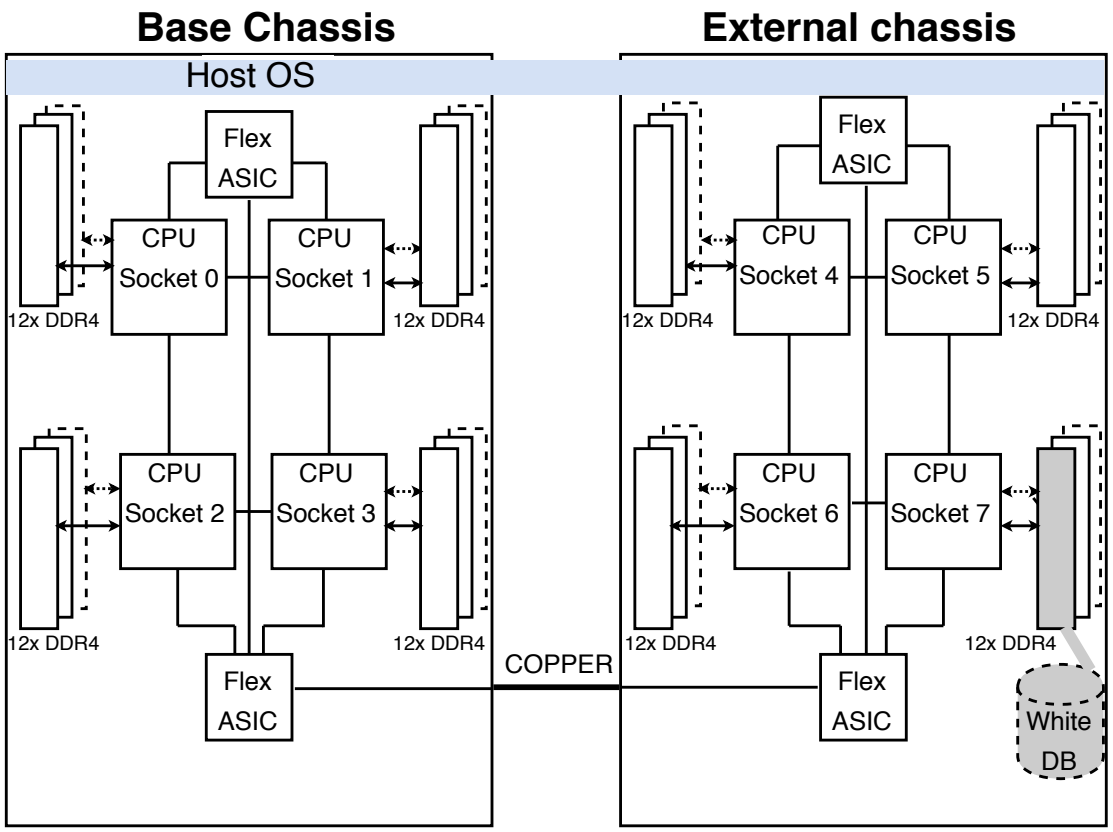


Figure 9: The memory-centric back-end surrogate used for the experiments

4.2 Measurement methods

The data access latencies are measured with equivalent techniques for both back-end technologies. The aim is to mimic data accessing in a VNFC environment as realistically as possible, including SDL functionality. An SDL API could provide this functionality, and it was available for the measurements. However, because of the differences in Redis and WhiteDB DBMSs, a specific simulation application was implemented for each technology, and there were slight differences in the underlying system instance for each technology. The basic application logic was kept the same: desired access operations were requested from the back-end and latencies of completed access operations were measured. The measures are eventually used to study the access performance.

4.2.1 Shared Data Layer API

The SDL API is a software component with which VNF applications gain the SDL functionalities described in Section 3.2. It is interoperable with different storage back-ends. It conceals the back-end specific data access details into object-oriented classes and provides abstract data manipulation methods to interact with a back-end available for the VNF application. Such manipulation method is for example read,

write or remove. To support multiple storage back-ends, SDL API is eventually planned to consist a class implementation for multiple back-ends. Then, the API could select the best suited back-end to serve particular data accesses made by a VNF application. By the time of the work, an implementation for the actual SDL API and a class implementation for Redis back-end had been developed by the client organization.

The API is imported to application program as a library which enables the program to instantiate SDL objects. To access the data records in a program, such SDL instance is called with an instance method corresponding to one of the mentioned manipulation methods. The manipulation methods offered by the API enable to perform simple one-time manipulation or read operations as well as more complex sequences of such operations. The operations read or manipulate records in the underlying storage. In terms of this work, simple SDL methods making multiple writes and reads of records to or from the database were used.

The API contains both synchronous and asynchronous interfaces for the instance methods. Synchronous and asynchronous methods differ in where the control flow of the calling program resumes when the method is called. In synchronous methods, the program blocks the proceeding until the I/O operations are processed, that is, the control flow proceeds to the next statement only when the operations are resolved. [53] The operations are dispatched one after another, such that the processing time of a method call is added in full to the execution time of the calling program. Asynchronous methods, instead, start the I/O operations that the method pertains but continue the execution of the calling program to the subsequent statements. [53] When the related I/O operations are complete, the calling program will invoke a callback function registered to that asynchronous method. Including asynchronous API calls to a program makes it an event-based program, as the program waits the I/O completion events and its proceeding depends on these events. [53] The event-based control takes place on a loop that continuously polls for possible I/O events issued by asynchronous API calls. Both synchronous and asynchronous data accesses are used in VNF applications, with the asynchronous accesses being more favored due to their performance benefits.

The API involves a namespace in every manipulation method call to associate an appropriate data type. A namespace is a grouping concept of the programming language: it determines a scope of names that are unique only inside this scope. By means of this, unique record identifiers, such as keys in the case of a key-value store back-end, can be defined in the scope of particular data type. These can overlap with record identifiers of other data types. For example, when an application has to process both subscription and charging data, a name *"id1"* can be used to express both a subscription data record and a charging data record in the application, given that a namespace for both types has been defined. Namespaces can be based on more fine-grained levels as well, such as on the logical parts of control plane or user plane functions.

The manipulation methods supplies VNFs and analytics applications with the ability to access the underlying back-ends. Assumably, VNF applications request mainly small-sized data blocks. Analytics applications might also request blocks

with a larger size. The data manipulation methods of the API can fulfill major of the properties specified for SDL. Therefore, the API is appropriate for simulating data accesses in cloud-native core network.

4.2.2 Test applications

Tests are conducted on a set of virtual system instances or processes. One test runner is harnessed as an orchestrator and the others as simulators. Runners are commonly termed as nodes within this section. For Redis tests, a separate system instance is set up to serve as an orchestrator: it manages launching and stopping of simulation applications at the simulator nodes as well as recovering them to desired states. For WhiteDB tests, the orchestrator node is an auxiliary process that just starts the actual simulation application processes with the settings given as parameters.

The Redis node instances might be either QEMU VMs or Docker containers. VMs virtualize a complete OS and hardware, whereas containers span just a group of user-space processes in an existing OS [47][66]. In both VM and container mode, a standard Linux distribution image is supplemented with a custom set of libraries and software required in the simulator functionality, resulting in a simulator image. An orchestrator image is prepared in a corresponding way except with a different software set. When a simulation is launched, instances of these orchestrator and simulator images start to come up. The simulator images are also given the test program executable file to be run when the orchestrator instructs them to do so. An instance in a running state i.e. a node can start to execute the test program. One of the nodes has to be dedicated as a database node, which solely creates a Redis database and runs the server that responds to access requests made by simulator nodes. The mentioned details are configured in Dockerfiles for container instances [68], and in OpenStack's Heat template files for VM instances [70]. The VM mode of Redis tests makes possible to have a physical network between the nodes. Although, all the Redis tests in this work were conducted with the container mode where nodes are lying at the same physical node and communicate via virtual network.

The Redis simulator nodes run a C++ -based simulator application. This application imports the SDL API with the ready-made Redis class implementation as such to its source code. In this way, Redis simulator applications can call the SDL API methods to either write records to or read records from the back-end. A desired simulation is configured with command line parameters given in the running of a simulator application executable. When a particular access pattern scenario is desired to be simulated for Redis, a few configuration files are edited to detail the launching of desired node composition, the networks between the nodes and the storage locations for measurement result files. Then, an orchestator node script is run with the edited configuration files passed as parameters. This first creates the system instances for the nodes and then start simulations on each of them. Parameters for the simulator programs are specified in the configuration files as well: programs obtain them via the passed configuration files.

WhiteDB nodes are C program processes which use the WhiteDB C API to create, attach and access a WhiteDB database [62, p. C API]. The specified number

of nodes are launched by the WhiteDB orchestrator node which takes the simulator program settings as arguments and passes them to each program. When a simulator node is writing to the WhiteDB database, the other concurrent simulator nodes cannot write until the one has finished. This is caused by the database locking which was used as the concurrency control mechanism in write experiments [62, p. Tutorial]. The command line arguments of both Redis and WhiteDB simulator programs were the same. They are provided in Table 1 below.

Table 1: Simulation application arguments

Argument	**Description**
numOps	number of access operations to be requested in a sequence
numLoops	the amount of repeats for access sequences
dbSize	size of the database to be attached by WhiteDB nodes
intervalTime	the time between each sequence when the sequence is requested more than once
recordSize	the size of record(s) being accessed
numProcs	the number of simulator nodes (writer/reader processes) to be spawned
accessMode	the access mode (read/write)

The settings of the simulators enable to test the performance of a back-end deployment with various access patterns, different amount of workload and concurrent clients. The simulation applications measure the average latency of a data access in each write/read sequence being requested from the back-end by each simulator node. The applications eventually produce output files where the average latencies and the total latency of the application is written. These files are used for used for post-analysis on the performance of each back-end. With both back-ends, the data to be stored to the database records is synthetic string data. This type of data could be present in a real VNF environment. Because WhiteDB was discovered to perform some optimizations to the size of the stored data when similar strings were written, the data used with WhiteDB was decided to be randomized.

Figure 10 depicts a general test setup. OrchApp refers to the orchestator application, which operates differently in the two back-end simulation setups. SimApp refers either to a Redis or WhiteDB simulation application: in case of Redis simulations, the SDL API is used to access PCN back-end, while in case of WhiteDB simulations, WhiteDB API is used to access MCN back-end. SDL API accesses the PCN back-end via TCP/IP connection while WhiteDB API accesses the MCN back-end (Linux shared memory) directly. The amount of simulator nodes is generally N, which is probably limited by the maximum number of instances that OpenStack or Docker Engine is able to handle.

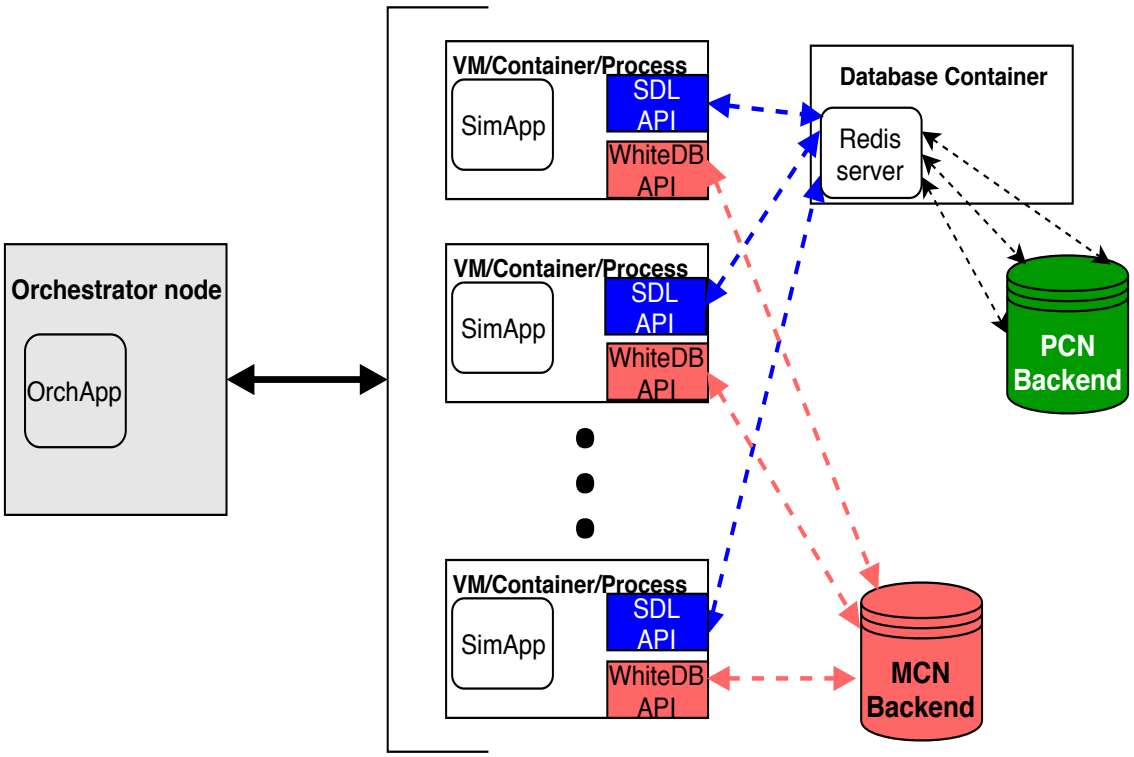


Figure 10: General test setup for data access simulations

5 Experiments

The first set of experiments is a comparison of processor-centric and memory-centric back-end latencies, and the second set of experiments is a proof of concept on the performance of accessing data via Gen-Z -type memory-semantic communication [39]. Data access simulations for both experiments are run on a Superdome Flex system described in Section 4.1.1. In the processor-centric back-end part of the first set, the hardware works in a same way to a processor-centric back-end, and a Redis database is set on top of that to serve the requested accesses. With this setup, Redis database is stored in the RAM of the same NUMA node where the simulation application is run. In the memory-centric back-end part of the first set, a WhiteDB database is configured to be stored in a Linux shared memory segment allocated from RAM of the same NUMA node where the application is run. In this way, the data was accessed from the local RAM similarly in both back-end setups. The second set of experiments considered the performance of direct load/store accesses across RAM paths on a memory-centric platform, due to which these experiments were performed only with shared memory -based WhiteDB databases. The location of WhiteDB databases was altered to test different accesses within the Superdome Flex system.

5.1 Access latency difference between back-end technologies

The latency is measured from the two back-ends by running multiple simulations on them. Multiple simulation configurations are produced by varying the node count, the access mode of the nodes (read/write) and the amount of repeated sequences. In Redis simulations, the Docker containers that run the simulation application are used as nodes, while in WhiteDB simulations, nodes are simulation application processes running on the native OS. Redis tests also involve a database node which runs a Redis server that serves the access requests made by the simulation applications at the other nodes. The relevant steps in setting up any viable simulation configuration are described in Section 4.2.2. The Redis database is stored in the local memory of the test system but the requests to access data happen via TCP/IP connections between each simulator node and the database node [34]. For memory-centric back-end measurements, a database of 300 GBs in size is first allocated with *wgdb* command-line utility [62, p. Tools]. An SHM segment of corresponding size is created on the background. Examples of actual commands for making WhiteDB operations are provided in Attachment A. A clean WhiteDB memory segment was allocated or new Redis database was created before each WhiteDB or Redis write simulation, respectively. Thus, all the write simulations are started with an empty database. Read access simulations are done on a database containing 1 kilobyte records. The different simulation configurations were decided with the help of the ideas presented by SDL experts: the 1 kilobyte record size was the maximum expected payload to be accessed by VNFs, while 100 000 operations was the maximum of total simultaneous operations. Dataset size of a simulation is the byte size of the set of records requested from the database at each arrival of an access sequence. The dataset size is defined

with the following equation, where the record size unit is bytes:

$$size_{dataset} = size_{sequence} size_{record} \quad (1)$$

Table 2 below shows the decided simulation configurations for both Redis and WhiteDB back-ends. The dataset sizes in the table are calculated according to Equation 1.

Table 2: Simulation configurations of first experiment set

#	**Simulation settings**	**Dataset size**	**Access mode**
1	Sequence size = 10000, Record size = 1 KB, Sequence repeats = 1, Nodes = 10	9.77 MB	Write
2	Sequence size = 10000, Record size = 1 KB, Sequence repeats = 1, Nodes = 10	9.77 MB	Read
3	Sequence size = 10000, Record size = 1 KB, Sequence repeats = 1, Nodes = 100	9.77 MB	Write
4	Sequence size = 10000, Record size = 1 KB, Sequence repeats = 1, Nodes = 100	9.77 MB	Read
5	Sequence size = 30000, Record size = 1 KB, Sequence repeats = 10, Nodes = 3	29.3 MB	Write
6	Sequence size = 30000, Record size = 1 KB, Sequence repeats = 10, Nodes = 3	29.3 MB	Read

5.2 The memory location impact on access latency in memory-centric back-end

The latency is measured on accesses to RAM of each separate NUMA node in the 8-socket Superdome Flex system. A WhiteDB database is created with "wgdb create" command before each write simulation [62, p. Tools]. The SHM segment size allocated for this database is 300 GBs in size, as in the first experiment set. The location of the RAM from which the segment for a WhiteDB database is allocated is controlled by binding the effective NUMA node by numactl tool [71]. Examples of using wgdb commands as such and within numactl are provided in Attachment A. Each of the eight nodes are used as the target for memory allocation at a time. The NUMA node 0 was fixed to act as the compute node such that the simulation applications were run in every test from that CPU socket. The simulation applications attach to the existing WhiteDB database by giving the same database identifier as given in earlier executed "wgdb create" command. Thus, the measurements are about accessing memory with different available memory bandwidths. All the write access simulations are started with an empty database. Read access simulations are done on a database containing sufficient amount of records of the specified size. The measurements were expected to exhibit the three different levels of latency specified for Superdome Flex. The setting values for the experiments were decided with the help of ideas presented by Superdome Flex experts. The amount of reader processes

for simulations was set to be 55, which is one less than the count of processor cores in a single NUMA node of Superdome Flex. The reasoning of this is that the computing capacity of the compute node would be used optimally when just about every reader process would have its own core to run on. To avoid caching effects at the compute node, every reader reads such a set of records from the database which are excluded from the record sets read by other readers. The sequence size parameter was set accordingly. Because WhiteDB has a locking mechanism to prevent doing simultaneous writes to the database, writers do not follow this practice. The simulation parameters is set up to a bit different values than in the first experiment set: one purpose of these changes is to make nodes to access such large datasets that the last-level cache of the compute node socket is certainly bypassed. Table 3 below shows all the simulation configurations that are run with each possible shared memory NUMA node option. The dataset sizes in the table are calculated according to Equation 1.

Table 3: Simulation configurations of second experiment set

#	Simulation settings	Dataset size	Access mode
1	Sequence size = 10000, Record size = 1 KB, Sequence repeats = 1, Nodes = 10	9.77 MB	Write
2	Sequence size = 1800, Record size = 1 KB, Sequence repeats = 1, Nodes = 10	1.76 MB	Read
3	Sequence size = 8000, Record size = 1 MB, Sequence repeats = 1, Nodes = 1	7.63 GB	Write
4	Sequence size = 1000, Record size = 1 MB, Sequence repeats = 1, Nodes = 55	0.95 GB	Read
5	Sequence size = 500, Record size = 1 MB, Sequence repeats = 2, Nodes = 55	0.48 GB	Read

6 Results and analysis

The results are based on average latencies to complete a single access, and a single average latency is determined by dividing the total latency of finishing a sequence of reads or writes by the sequence size. The variances of all the experiments included in the two sets are summarized with arithmetic mean, and the spread is summarized with standard deviation. The arithmetic mean is calculated with the following well-known equation:

$$\bar{x} = \frac{1}{n}\sum_{1}^{n} x_i \tag{2}$$

Standard deviation is, in turn, defined with the well-known equation:

$$s = \sqrt{\sum_{i=1}^{N} \frac{1}{N} (x_i - \bar{x})^2} \tag{3}$$

In the following sections, the results of the experiment sets for "Access latency difference between back-end technologies" and "The memory location impact on access latency in memory-centric back-end" are presented.

6.1 Access latency difference between back-end technologies

6.1.1 Results

The order of the latencies of the two back-ends are visualized in histograms. In each such histogram, the vertical axis shows the frequency of a latency range occurred in the latency measurements. The frequency of each latency range is shown as a percentage of the total number of latency measurements for each back-end technology. The horizontal axes shows access latency in either microseconds or nanoseconds (ns) in a linear scale. The mentioned statistical measures are given for each experiment in Table 4 to provide more insights on the average latency and spread.

Table 4: Statistical measures of the experiments of the first experiment set

#	(Mean [ns], Standard deviation [ns]) in Redis	(Mean [ns], Standard deviation [ns]) in WhiteDB
1	(9468, 1701)	(4620, 466.4)
2	(36770, 1457)	(19.8, 4.094)
3	(97230, 46140)	(5020, 852.8)
4	(375400, 2215)	(44.68, 58.82)
5	(35550, 4899)	(2200, 568.9)
6	(97055, 19507)	(39.07, 32.67)

Experiments 1 and 2 were simulations with parameters that can be thought to occur in a real VNF environment at busy times. The record size and simultaneous operations were in the maximum order expected from SDL use cases. Figure 11 shows

the Redis and WhiteDB latencies of Experiment 1 in the same histogram. Figure 12 shows the latencies of Experiment 2 in separate histograms for the back-ends placed side by side. The time unit is microsecond in Redis latencies, while nanoseconds are used in WhiteDB latencies.

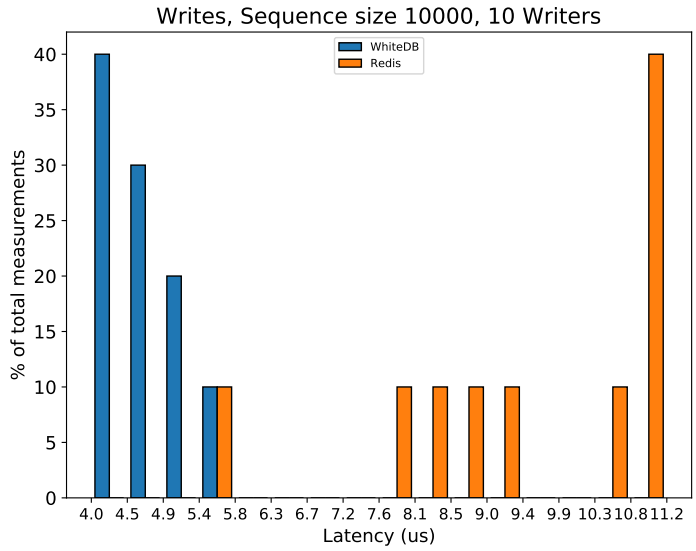
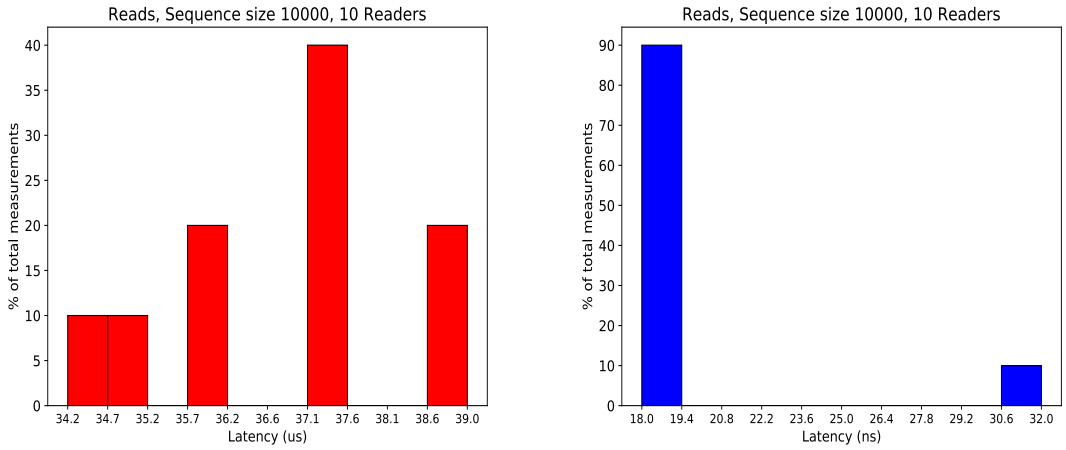


Figure 11: Write latencies occurred in Experiment 1 for each back-end

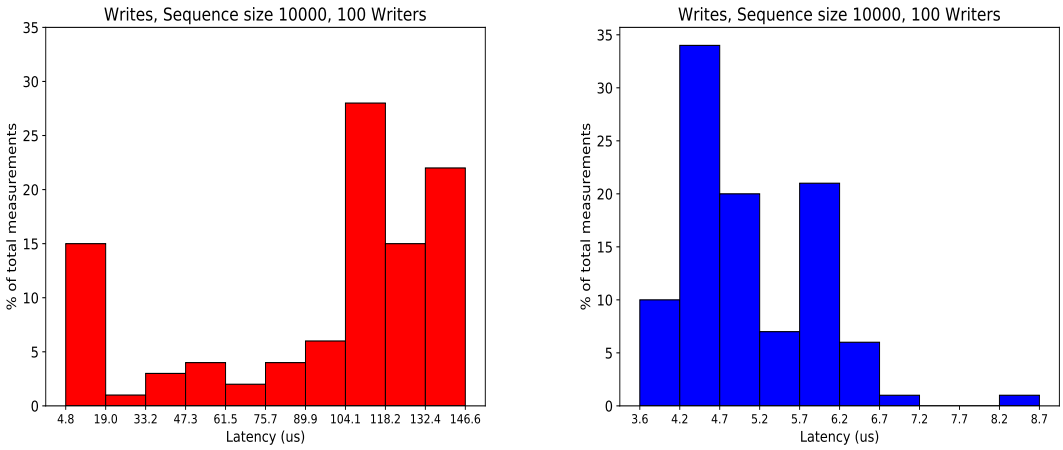


(a) (b)

Figure 12: Read latencies occurred in the Redis accesses of Experiment 2 (a) and corresponding latencies of WhiteDB accesses (b)

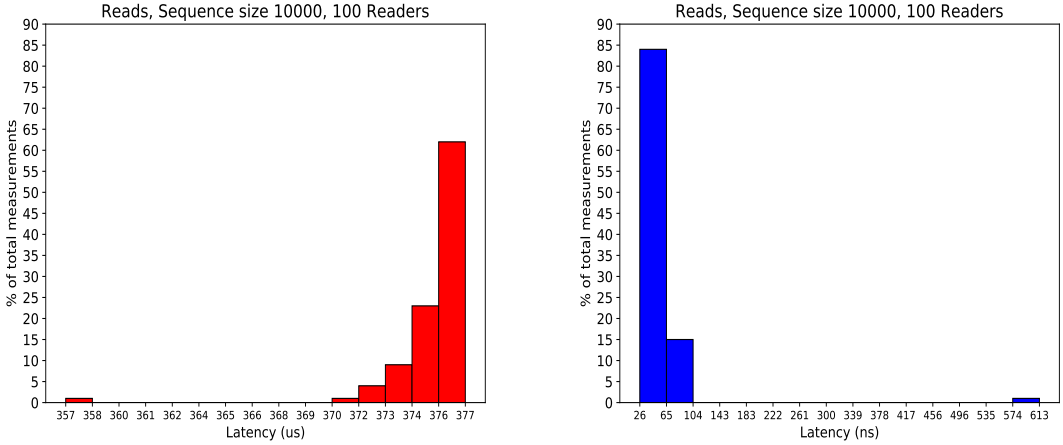
Experiments 3 and 4 were performed to try to see how a large number of nodes accessing the back-end simultaneously affects the performance. The other parameters

were set to same as in Experiments 1 and 2, and so the Experiments 3 and 4 were as extreme cases to see how the back-end performs in unexceptional load conditions. Figures 13 and 14 show the latency histograms drawn from Experiments 3 and 4, respectively. The latencies are shown in separate histogram for each back-end. In Figure 14, the time unit used in the Redis histogram is microsecond, while nanoseconds are used in the WhiteDB histogram.



(a) (b)

Figure 13: Write latencies occurred in Redis accesses of Experiment 3 (a) and corresponding latencies of WhiteDB accesses (b)

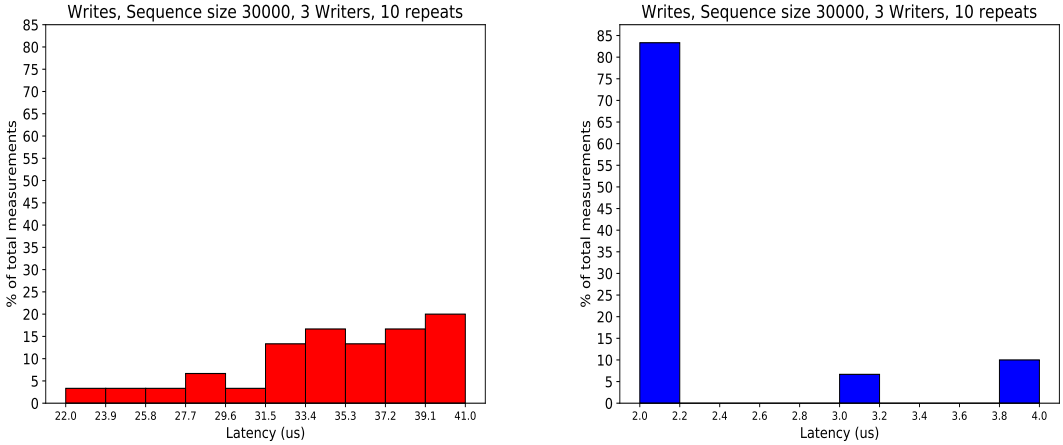


(a) (b)

Figure 14: Read latencies occurred in Redis accesses of Experiment 4 (a) and corresponding latencies of WhiteDB accesses (b)

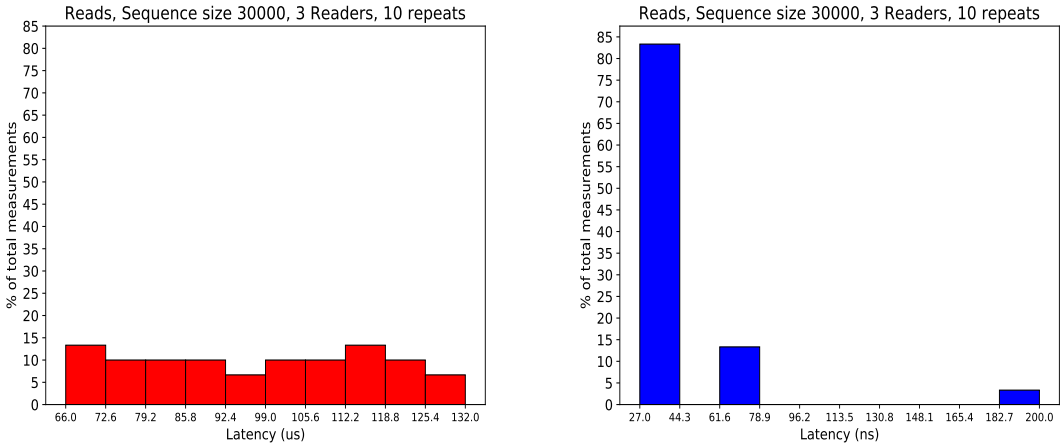
Experiments 5 and 6 were performed to try see how repetition of sequences affects on the access performance of a back-end. Each node were instructed to repeat a

sequence of accesses 10 times. The sequence size was adjusted to 30 000 and the node count to 3, such that the total amount of simultaneous operations was in the same order as in Experiments 1 and 2. Figures 15 and 16 show the latency histograms drawn from Experiments 5 and 6, respectively. The latencies are shown in separate histogram for each back-end. In Figure 16, the time unit used in the Redis histogram is microsecond, while nanoseconds are used in the WhiteDB histogram.



(a) (b)

Figure 15: Write latencies occurred in Redis accesses of Experiment 5 (a) and corresponding latencies of WhiteDB accesses (b)



(a) (b)

Figure 16: Read latencies occurred in Redis accesses of Experiment 6 (a) and corresponding latencies of WhiteDB accesses (b)

6.1.2 Analysis

Figure 11 summarizes Experiment 1 and Figure 12 summarizes Experiment 2. Figure 11 shows that 70 % of the writes to WhiteDB were in the range 4-4.7 microseconds (us) and 30 % of the writes were in the range 4.95-5.6 us. 40 % of Redis writes were in the range 11-11.2 us and the rest of them spread to the range of 5.6-10.75 us. The arithmetic mean of (9.47 us) and standard deviation (1.70 us) of Redis writes implies that the latencies were concentrated close to 10 us. The corresponding values for WhiteDB were 4.62 us and 0.47 us. These values affirm that WhiteDB writes were faster in average and that their variation were only one third of that of Redis. Figure 12 shows that read latencies were in totally different ranges between the back-ends: all the Redis latencies fall within 34.2-39 us range whereas all the WhiteDB latencies fall within 18-32 ns range. The arithmetic means for Redis and WhiteDB reads were roughly 37 us and 18 ns, respectively. The figure and standard deviations in Table 4 show that variation was relatively higher in WhiteDB than in Redis reads. Although, the absolute variation was still clearly higher in Redis than in WhiteDB.

As Figure 13 shows, Redis latencies tend to increase when an exceptionally big number of nodes are accessing sequences of the same size as in Experiments 1-2. Figure 13a details that 85 % of the Redis writes are taking longer than the longest WhiteDB write. About 65 % of Redis writes lasted 104-147 us and the remaining portion spread from 5 to 104 us. The big increase in node count did not affect to WhiteDB write latencies that much: the ranges were similar and median was almost equal to the arithmetic mean in Experiment 1. Locking of the complete database for writes is probably a partial cause for this, as a bunch of 10 000 writes from each node is served one after the other. The WhiteDB latencies lasted up to 8.7 us and thus the relative variation of Redis latencies was much higher. Figure 14 shows that Redis read latencies increased along with the increase in node count: 99 % of the latencies were in range 370-377 us which is about tenfold to the latencies measured in Experiment 2. 84 % of WhiteDB reads were in 26-65 ns range and most of the remaining latencies lasted from 65 to 104 ns. The relative variation of WhiteDB reads was higher than of Redis reads. One WhiteDB latency was in the 574-613 ns range which can not be explained comprehensively; one possibility might be a higher load in the OS during the short period of read.

Figure 15a shows that Redis writes tend to take longer when the sequence size becomes tripled and requesting of the sequence is repeated 10 times in a row from each node. Figure 15b shows that all the WhiteDB write latencies were in 2-4 us range, which are the fastest WhiteDB writes from all the write experiments. As 84 % of them were in 2-2.2 us range and the standard deviation was only about 0.56 us, repeating of access sequence requests did not cause much higher latency. The mean for Redis write was 35.6 us but the latencies were quite widely spread between 22 and 41 us. Figure 15b shows that Redis reads followed the same trend as Redis writes in Experiment 5. WhiteDB reads had a slightly lower spread than in Experiment 4 with 100 nodes but generally the latency pattern was very similar to that. Table 4 shows that a Redis read took roughly 100 us and a WhiteDB read roughly 40 us by average. Overall, the experiment showed that Redis back-end could not cope with

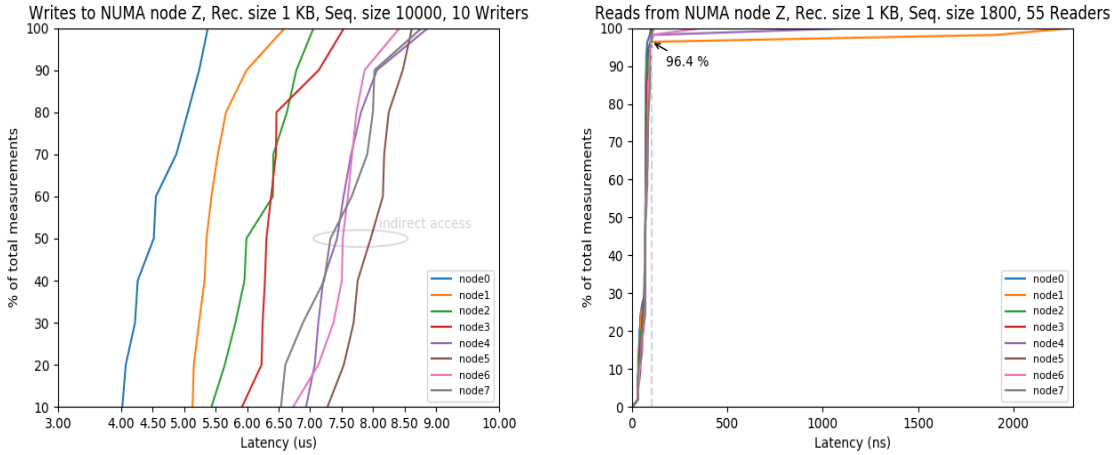
consecutive requests of bunches of accesses.

In summary, the back-end comparison results show that TCP/IP communication involved in Redis accesses causes higher latencies than direct accesses to RAM done with WhiteDB. The increase is very distinct in read accesses as Redis latencies were at least three orders of magnitude higher than WhiteDB latencies. The extreme read Experiments 4 and 6 imply that WhiteDB performance scales well with exceptional loads while Redis latencies became multiplied when the load was increased to extremes. In write latencies, WhiteDB outperformed Redis with a smaller margin than in reads, nevertheless, TCP/IP caused overhead is visible also in their comparison. The results showed that WhiteDB write latencies has lower relative variation than Redis write latencies, but in read latencies WhiteDB was relatively more variable than Redis. As the latencies are averages over total latencies of large access sequences, variation can not be evaluated as firmly as the latency magnitudes. As error consideration for the results, the Docker Engine used to run Redis container nodes could have caused some of the overhead and variation in Redis latencies. Only WhiteDB read accesses were truly happening simultaneously to the shared memory, because of which they can be thought to be closer to accesses to the databases that will be happening in The Machine.

6.2 The memory location impact on access latency in memory-centric back-end

6.2.1 Results

The results of each NUMA node access experiment detailed in Table 3 at "Experiments" section are illustrated here. For each experiment, an empirical cumulative distribution function (ECDF) with probability normalized to percentage is calculated for each of the eight NUMA node memory accesses. The memory accesses are described in Section 5.2. The ECDFs visualize central tendency and spread of the latencies for all the Experiments except Experiment 3: for this experiment, a single latency for accessing the memory of a specific NUMA node is given in a table. The horizontal axes of the ECDFs show the access latency either in microseconds or nanoseconds in a linear scale. Figure 17 shows the latencies of accessing memory of a certain NUMA node from the compute node (NUMA node 0), with a 1 KB record size.



(a) (b)

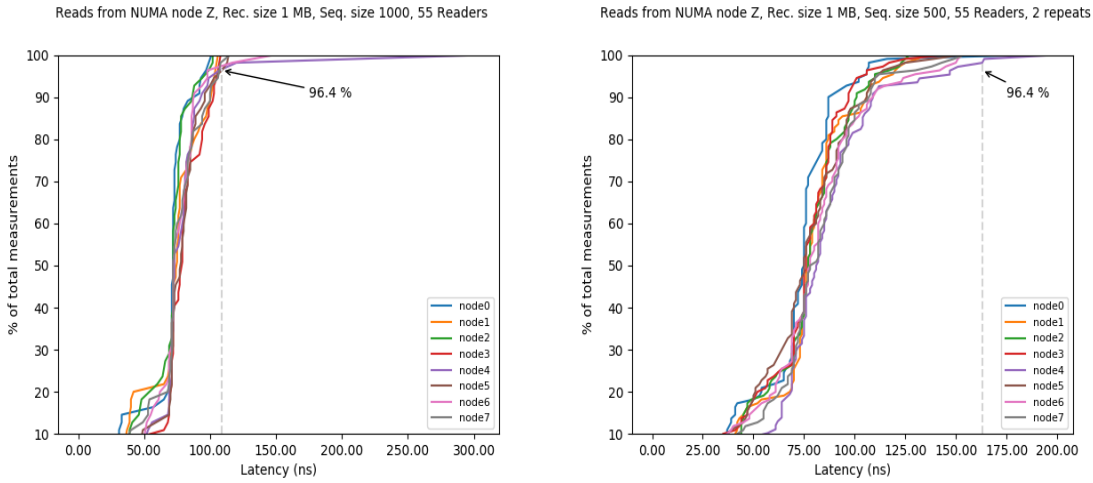
Figure 17: Access latencies occurred in Experiment 1 (a) and Experiment 2 (b), Experiment 1 was write accesses and Experiment 2 was read accesses

In Experiment 3, the sequence size was increased to 1 Megabyte (MB) to ensure that memory accesses would really happen from the RAM of the NUMA nodes. One simulator node was configured to do a sequence of writes of size 7.63 GBs. Experiment 3 yielded just one sample for each memory access measurements, and the results are thus shown in the Table 5 below.

Table 5: Access latency in Experiment 3

NUMA node #	Access latency (ms)
0	2.032
1	2.227
2	2.226
3	2.342
4	2.447
5	2.455
6	2.568
7	2.434

In Experiment 4, 1 MB records were read from the database. The sequence size was set up such that each node read a certain subset of the records in the database, with the intention that caching would not be involved in the process. This size was 1000. Experiment 5 was otherwise similar to Experiment 4 except that the sequence size was halved to 500 and the reading of this sequence was repeated once. The results of Experiments 4 and 5 are shown in Figure 18.



(a) (b)

Figure 18: Access latencies occurred in Experiment 4 (a) and Experiment 5 (b), both experiments were read accesses

6.2.2 Analysis

Figure 17a shows that writes to any node throughout the 8-socket Superdome Flex fabric have latencies of the same order. The curves of the nodes located at the external chassis are marked with "indirect access" note. The curves show that indirect accesses to memory (NUMA nodes 4-7) were consistently a bit slower than direct accesses (NUMA nodes 0-3). Particularly, local memory (NUMA node 0) was the fastest to access. Difference in the medians of the fastest NUMA node 0 and slowest NUMA node 5 access is approximately 3.4 us, which is a notable size. The nodes 0-3 were located at internal chassis and nodes 4-7 at external chassis. Thus, the three different latency levels of Superdome system can be seen in 1 KB writes. Figure 17b shows that the location of the memory throughout the system does impact very little or nothing to read latency. 96.4 % of all the reads lasted about 100 ns at most. A few sequences accessed from NUMA node 1 and NUMA node 4 fall within microsecond range average latency.

Table 5 shows write latencies of different NUMA node accesses across the platform. The latencies were generally of the same order. Similarly to Experiment 1, locality of the memory was visible in the latencies as internal memory accesses were faster than external, and the local memory access was the fastest among internal accesses. Although the 1 MB record size caused inappropriately high latencies, the results showed that the location of memory did not radically affect write latency in Superdome Flex.

Figure 18a indicates that all the accesses to memory of NUMA node 0 completed in about 97 ns at most, and secondly, all the accesses to the memories of NUMA node 7 or 5 completed in roughly 110 ns at most. This implies that the location of NUMA node from where memory is accessed does not impact latency in 1 MB record reads. A few sequences lasted 150-300 ns in average, but 96.4 % of all the reads lasted at

most 110 ns in average. Figure 18b indicates basically the same result: latencies were pretty equal regardless of memory location. As a detail of both experiments, accesses to the NUMA node 4 at the external chassis seemed to take the longest times. This is in line with that latency of indirect accesses is higher than of direct accesses, but it can be also just a statistical bias.

To summarize, direct load/store accesses showed that latencies remain in the same order when accesses are taking different paths across memory-semantic fabric. Virtually all the write and read latencies were in applicable levels for VNF applications. The different latency levels of Superdome Flex are visible in write latency comparison, but read latencies showed very little differences in accesses to different locations over the fabric. The database was locked for writes but freely read, which might be a major reason for this. Furthermore, 1 KB and 1 MB record reads showed similar latency levels, which probably indicates that some factor of the environment was not taken into account. Such possible error factors could be that the last-level cache was not bypassed in all the accesses or that the WhiteDB software has some hidden data optimization features. The variations in access latencies to a specific NUMA node remained in tolerable sizes also. The few microsecond scale latencies in Experiment 2 were probably caused by measurement errors or an external factor.

7 Conclusions

In this work, memory-centric network architecture was explored for its suitability in SDL storage back-end use, with focus on ultra-low latency services. Memory-centric network builds essentially on fabric-attached memory, Gen-Z system interconnect and non-volatile memory. One of the most extensive researchers around the topic has been HPE: they have developed a comprehensive The Machine platform, as well as other platforms that utilize memory-driven computing in a smaller scale. One smaller scale platform, Superdome Flex, was obtained for the experiments of this work. The performance impacts of memory-centric technologies were studied by running two experiment sets on this platform. First set of experiments concerned latency differences between conventional TCP/IP -based back-end accesses and shared memory -based back-end accesses. Second set of experiments concerned latencies among different memory access paths in the Superdome Flex platform. In addition to latency differences, consistency, availability and reliability of the old and new back-end architecture were compared qualitatively.

The TCP/IP-based accesses were performed via Redis database, and the shared memory -based back-end accesses via WhiteDB database. The database was in both cases living in the same OS and the memory accesses were done to the local memory of the "node", that is, a CPU socket of Superdome Flex. The back-end comparison results will so apply to a system corresponding to a Superdome Flex with a single OS where data is accessed either via Redis or WhiteDB DBMSs. WhiteDB setup showed much better performance than Redis setup in reads, with WhiteDB latencies being in nanosecond range. In writes, the latency differences were slighter but WhiteDB writes were still consistently faster than Redis writes. These results say that the TCP/IP -stack causes a lot of overhead to memory accesses. The write latency results reflect the similar write concurrency control i.e. an exclusive access for each write, of the back-ends. When a certain degree of consistency is required from VNF application, a traditional back-end, like the tested Redis setup, seems still more suitable. This is because the studied memory-centric databases offer only an absolute consistency (WhiteDB) or depend on immature non-volatile memory (FOEDUS). Availability and reliability of memory-centric back-ends are also hard to judge, as VNF nodes require modifications in the OS to implement memory management and failure handling of fabric-attached memory properly.

The access path experiments were performed with a setup where the location of a WhiteDB database was varied across the platform. The location was found to not specially affect on the latency that an access took. Write latencies were rising when the WhiteDB database was placed to the RAM of a NUMA node farther and farther away from the compute node, but the magnitude stayed the same. With reads, the latencies were at the same level through any memory access path within the memory-centric back-end. When the record size was changed to excessively high for SDL use, the write latencies grew to milliseconds but read latencies stayed at the same level. The total absence of concurrency control in read accesses is very likely one major reason for this effect.

Traditional storage systems are adding a significant amount of overhead to

memory accesses because of TCP/IP communication. The latency of accessing data from such systems is likely to be too severe for the desired latency-critical services. Memory-centric platforms provide a promising alternative for a SDL back-end since its memory-semantic communication protocol causes very little overhead to accesses. Although the memory-centric setup used in the experiments did not include all the components and characteristics of The Machine prototype platform, it showed the reduction in latency that a complete back-end system of this type could achieve in relation to currently used back-ends.

7.1 Applicability on 5G use cases

The roughly uniform latency along any access path of a back-end could be utilized in several planned use cases. One such use case from IoT area is self-driving cars, which is considered to be possible with 5G communication [1, pp. 376-377]. Self-driving cars will have a necessary requirement for maximum 1 millisecond latency in car-to-radio communication [4]. The data processing related to adjacent roadside radios will be likely performed in the same edge cloud, which advocates to employ memory-centric back-end as the storage system for such an edge cloud. As the memory location impact experiments showed, any RAM of a memory-centric back-end with two attached compute nodes can be accessed with equal latency, meaning that a common edge cloud could benefit from the back-end in the case of two adjacent cells serving a self-driving car. This result can be thought to hold also for memory-centric back-ends with up to 8 fabric-attached nodes, because the developer has promised the performance to scale up to 8 nodes for the back-end type used in experiments [59].

A memory-centric back-end SDL deployment would also cut the need to have a local copy of a database object at every microservice that a VNF consists. The microservices could be spread to the SOCs of a memory-centric platform, such that each microservice would be run on separate cores independently from other services. Then, each microservice could access any RAM throughout the platform with roughly equal latency, and thus database objects can be accessed just on demand from their location. The experiments were made in a Superdome Flex platform with 8 processor sockets or SOCs, which allows a VNF with reasonably high performance demand to be hosted.

Increasing of accessor nodes from 10 to 100 did not seem to affect on access latency in the tested memory-centric back-end deployment. Such independency could be useful in a use case where the network has to process just established mobile sessions together with already ongoing mobile sessions. A network is said to be able to do non-intrusive processing when the processing time of the incoming mobile sessions is not affected by the queue of ongoing mobile sessions [1, p. 290]. The results promise that non-intrusive processing could be possible in a mobile network with a memory-centric storage back-end deployed.

7.2 Future work

Available memory-centric setups, such as WhiteDB or FAME on top of Superdome hardware, would be important to be studied for consistency, reliability and availability together with the performance. For example, a database with more subtle locking would be interesting to measure and compare with the latency results obtained in this work. In addition, if the quality characteristics could be evaluated quantitatively, the suitability of memory-centric platform in SDL use could be judged better.

Memory-centric back-end systems need to be studied with more comprehensive experiments to quantify the latency factors of each involved hardware and software components. Such components can be the API to the back-end database, the database program and concurrency control mechanisms of the OS. When tests are able to separate the latency caused by system interconnect level components, one might be able to model the access latency of the memory-centric platform analytically. The platform might be for example Superdome Flex.

The networking of a memory-centric fabric or fabric-attached nodes is important to be studied. This work discussed very concisely some network topologies and routing algorithms that might suit for a memory-centric back-end. The different topologies should be measured and evaluated on their performance and other qualities. The possible topologies with which the chassis of a Superdome Flex can be interconnected might be one interesting study area.

8 References

[1] M. K. Weldon, *The Future X Network : a Bell Labs Perspective*. Boca Raton: CRC Press, 2016.

[2] V. Ziegler and T. Theimer and C. Sartori and J. Prade and N. Sprecher and K. Albal and A. Bedekar, "Architecture vision for the 5G era," in *2016 IEEE International Conference on Communications Workshops, ICC 2016*, pp. 51–56, 2016.

[3] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and Research Challenges," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[4] Nokia Networks, "Building a cloud-native core for a 5G world," 2017. Strategic White Paper. Online. Accessed 06.03.2018. https://onestore.nokia.com/asset/200888/Nokia_AirGile_Cloud-native_Core_White_Paper_EN.pdf.

[5] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 145–155, Sept 2013.

[6] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond processor-centric operating systems," in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2015.

[7] K. Keeton, "Memory-Driven Computing," in *FAST*, 2017. Online. Accessed 09.12.2017. https://www.usenix.org/sites/default/files/conference/protected-files/fast17_slides_keeton.pdf.

[8] Nokia Networks, "Creating a new data freedom with the Shared Data Layer," 2016. White Paper. Online. Accessed 11.11.2017. https://onestore.nokia.com/asset/200238.

[9] M. Schlansker, J. Tourrilhes, S. Banerjee, and P. Sharma, "Network Function Virtualization and Messaging for Non-Coherent Shared Memory Multiprocessors," 2016. Hewlett Packard Labs.

[10] F.Chen, M. T. Gonzalez, K.Viswanathan, Q.Cai, H.Laffite, J.Rivera, A.Mitchell, S.Singhal, "Billion node graph inference: iterative processing on The Machine," 2016. Hewlett Packard Labs.

[11] M. Kerrisk, *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press, 2010.

[12] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[13] D. C. Marinescu, *Cloud Computing : Theory and Practice*. San Francisco: Elsevier Science, 20130524 2013. ID: 1213925.

[14] M. Fowler, "Microservices." Online. Accessed 05.01.2018. https://martinfowler.com/articles/microservices.html.

[15] Network Functions Virtualisation (NFV), "Infrastructure Overview," tech. rep., ETSI, 2015. Sophia Antipolis, France.

[16] Network Functions Virtualisation (NFV), "Infrastructure; Compute Domain," tech. rep., ETSI, 2014. Sophia Antipolis, France.

[17] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," in *International Symposium on Memory Management, ISMM*, pp. 11–20, 2011.

[18] J. Hennessy, M. Heinrich, and A. Gupta, "Cache-coherent distributed shared memory: Perspectives on its development and future challenges," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 418–429, 1999.

[19] Gen-Z Consortium, "About Gen-Z | Gen-Z Consortium | Open-Systems Interconnect." http://genzconsortium.org/about/.

[20] M. Alicherry and T. V. Lakshman, "Network aware resource allocation in distributed clouds," in *Proceedings - IEEE INFOCOM*, pp. 963–971, 2012.

[21] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Computer Communication Review*, vol. 38, pp. 63–74, 2008.

[22] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: Topology, routing, and packaging of efficient large-scale networks," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, 2009.

[23] E. Chow, "A Course on High Performance Scientific Computing on Intel Processors - Interconnects." Georgia Tech. University Course Material. Online. Accessed 27.12.2017. http://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/18_interconnects.pdf.

[24] Nitin and D. S. Chauhan, "Comparative analysis of traffic patterns on k-ary n-tree using adaptive algorithms based on burton normal form," *Journal of Supercomputing*, vol. 59, no. 2, pp. 569–588, 2012. Cited By :21.

[25] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[26] International Organization for Standardization, *ISO 8402: 1994: Quality Management and Quality Assurance-Vocabulary*. International Organization for Standardization, 1994.

[27] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, *et al.*, "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory," in *FAST*, vol. 11, pp. 61–75, 2011.

[28] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, vol. 151, ch. Introduction to Paging. Arpaci-Dusseau Books Wisconsin, 2014. Online. Accessed 06.03.2018. http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf.

[29] M. Kerrisk, *Linux Programmer's manual - MMAP(2)*. Online. Accessed 07.03.2018. "http://man7.org/linux/man-pages/man2/mmap.2.html.

[30] M. Kerrisk, "POSIX Shared Memory." Linux/UNIX System Programming Training Course 2017 Material. Online. Accessed 28.02.2018. http://man7.org/training/download/lusp_pshm_slides.pdf.

[31] E. L. Bosworth, *Computer Architecture and Design course (CPSC 5155) - Textbook*, ch. Memory Organization and Addressing. TSYS School of Computer Science, Columbus State University, 2011. Online. Accessed 21.02.2018. http://www.edwardbosworth.com/My5155Text_V07_PDF/MyText5155_Ch09_V07.pdf.

[32] E. L. Bosworth, *Computer Architecture and Design course (CPSC 5155) - Slides*, ch. Cache Coherency. TSYS School of Computer Science, Columbus State University, 2011. Online. Accessed 21.02.2018. http://www.edwardbosworth.com/My5155_Slides/Chapter13/CacheCoherency.htm.

[33] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *IEEE Symposium on Mass Storage Systems and Technologies*, vol. 2015-August, 2015.

[34] R. Cattell, "Scalable SQL and NoSQL data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2010.

[35] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 385–395, 2010. Cited By :125.

[36] A. M. Caulfield, J. Coburn, T. I. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the Impact of Emerging Non-volatile Memories on High-performance, IO-intensive Computing," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, 2010.

[37] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4-5, pp. 465–479, 2008. Cited By :439.

[38] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, "Spin-dependent phenomena and their implementation in spintronic devices," in *International Symposium on VLSI Technology, Systems, and Applications, Proceedings*, pp. 70–71, 2008. Cited By :14.

[39] Gen-Z Consortium, "Gen-Z: Core Specification," tech. rep., Gen-Z, February, 2018.

[40] S. M. Nowick and M. Singh, "High-Performance Asynchronous Pipelines: An Overview," *IEEE Design Test of Computers*, vol. 28, pp. 8–22, Sept 2011.

[41] M. Tan, P. Rosenberg, W. Sorin, S. Mathai, G. Panotopoulos, and G. Rankin, "Universal photonic interconnect for data centers," in *2017 Optical Fiber Communications Conference and Exhibition (OFC)*, pp. 1–3, 2017. ID: 1.

[42] I. Djordjevic, W. Ryan, and B. Vasic, *Coding for Optical Channels*. Boston, MA: Springer US, 2010.

[43] A. Singh, *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.

[44] S. Sesia, I. Toufik, and M. Baker, *LTE - The UMTS Long Term Evolution: From Theory to Practice: Second Edition*, ch. Network Architecture, pp. 25–55. LTE - The UMTS Long Term Evolution: From Theory to Practice: Second Edition, Wiley & Sons Ltd., 2011.

[45] 3GPP TS 23.501, "System Architecture for the 5G System," V15.0.0, 3rd Generation Partnership Project, dec 2017.

[46] Nokia, "Nokia CloudBand Infrastructure Software - Release 19," 2018. Online. Accessed 12.9.2018. https://onestore.nokia.com/asset/200058.

[47] QEMU under GNU Free Documentation License 1.2, *QEMU Wiki Page*. Online, accessed 14.01.2018 https://wiki.qemu.org/Main_Page.

[48] Nokia Networks, "Nokia CloudBand Application Manager - Release 18.x," 2018. Online. Accessed 12.9.2018. https://onestore.nokia.com/asset/200057.

[49] Nokia Networks, "Evolution of the cloud-native mobile core," 2017. White Paper.

[50] N. Networks, "Anatomy of a microservice," 2018. Online. Accessed 11.03.2018. https://onestore.nokia.com/asset/201751/Nokia_Microservice_Anatomy_White_Paper_EN.pdf.

[51] ETSI millimetre Wave Transmission (mWT) Group , "Applications and use cases of Software Defined Networking (SDN) as related to microwave and millimetre wave transmission," tech. rep., ETSI, 2017. Sophia Antipolis, France.

[52] H. Kimura, "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. 2015-May, pp. 691–706, 2015.

[53] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10*, pp. 186–189, 2002.

[54] S. Thongprasit, V. Visoottiviseth, and R. Takano, "Toward Fast and Scalable Key-Value Stores Based on User Space TCP/IP Stack," in *Asian Internet Engineering Conference, AINTEC 2015*, pp. 40–47, 2015.

[55] D. Cotroneo, L. D. Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network Function Virtualization: Challenges and Directions for Reliability Assurance," in *Proceedings - IEEE 25th International Symposium on Software Reliability Engineering Workshops, ISSREW 2014*, pp. 37–42, 2014.

[56] The Linux Foundation Projects, "DPDK: Data Plane Development Kit." Online. Accessed 18.03.2018. http://dpdk.org.

[57] D. R. Chakrabarti, H. J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pp. 433–452, 2014.

[58] T. C. H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017*, pp. 468–482, 2017.

[59] Hewlett Packard Enterprise, "HPE Superdome Flex server architecture and RAS," 2017. Technical white paper. Online. Accessed 03.06.2018. https://h20195.www2.hpe.com/V2/getpdf.aspx/A00036491ENW.pdf?

[60] Hewlett Packard Enterprise, "Fabric Attached Memory GitHub | Fabric Attached Memory Emulation repository | Project README." Online. Accessed 22.02.2018. https://github.com/FabricAttachedMemory/Emulation.

[61] N. McDonald, "Hewlett Packard GitHub | SuperSim repository | Project README." Online. Accessed 28.02.2018. https://github.com/HewlettPackard/supersim.

[62] T. Tammet and P. Järv, "WhiteDB - Main Page." Online. Accessed 9.4.2018. http://whitedb.org/index.html.

[63] Intel Corporation, “Intel Xeon Scalable Platform,” 2017. Product Brief. Online. Accessed 06.06.2018. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-scalable-platform-brief.pdf.

[64] R. Craig and T. Potter, “Fabric Attached Memory GitHub | Fabric Attached Memory Emulation repository | Emulation via Virtual Machines.” Hewlett Packard Enterprise. Online. Accessed 23.02.2018. https://github.com/FabricAttachedMemory/Emulation/wiki/Emulation-via-Virtual-Machines.

[65] P. Järv and T. Tammet, “WhiteDB GitHub Repository | Project README.” Online. Accessed 12.04.2018. https://github.com/priitj/whitedb.

[66] Docker Inc., *What Is A Container | Docker*. Online. Accessed 14.01.2018. https://www.docker.com/what-container#/virtual_machines.

[67] Docker Inc., *About Docker Engine*, 2018. Online. Accessed 14.04.2018. https://docs.docker.com/engine/.

[68] Docker Inc., *Dockerfile Reference | Docker Documentation*. Online. Accessed 17.02.2018. https://docs.docker.com/engine/reference/builder/.

[69] N. McDonald, “Hewlett Packard GitHub | SuperSim repository | Documentation.” Online. Accessed 11.03.2018. https://github.com/HewlettPackard/supersim/blob/master/docs.

[70] OpenStack Foundation, *OpenStack Docs | Heat Orchestration Template (HOT) Guide*, 2017. Online. Accessed 17.01.2018. https://docs.openstack.org/heat/latest/template_guide/hot_guide.html.

[71] A. Kleen, *Linux Man Page - NUMACTL(8)*. Online. Accessed 04.06.2018. "https://linux.die.net/man/8/numactl.

A WhiteDB pre-settings and database managing

A.1 Pre-settings

```
git clone https://github.com/priitj/whitedb.git
cd whitedb
./Bootstrap
./configure
make
make install
export LD_LIBRARY_PATH=$PWD/Main/.libs:$LD_LIBRARY_PATH
```

After the above commands, WhiteDB library can be linked to a C application in compilation (demo.c from WhiteDB project used as an example)

```
gcc -o Examples/demo Examples/demo.c -lwgdb
cd Examples
./demo
```

A.2 Database managing

```
# API call to create a DB of given size/attach to an existing DB if shmkey exists
wg_attach_database(char* shmkey, int size)

# API call to create a new record with a given length to the given DB
wg_create_record(void* db, wg_int length)

# API call to get a pointer that is next from the given record in the DB
wg_get_next_record(void* db, void* record)

# API call to set one record field to data in the given record lying in the DB
wg_set_field(void* db, void* record, wg_int fieldnr, wg_int data)

# create a 1 GB WhiteDB database bound to SHM key "1234" with wgdb utility
wgdb 1234 create 1000000000

# remove the database bound by "1234" and free the memory it reserved
wgdb 1234 free

# force that wgdb is run at NUMA node 0 and shared memory for the DB is allocated
# from RAM of NUMA node 4
numactl -m 4 -N 0 wgdb 1234 create 1000000000

# run simApp executable similarly to above i.e. only RAM of NUMA node 4 is used
numactl -m 4 -N 0 ./<name_of_simapp_executable>
```

B SuperSim pre-settings for running simulations

B.1 Installing required packages

```
apt-get install g++ git python3 python3-pip python3-dev python3-tk
sudo apt-get install libtclap-dev zlib1g-dev wget
pip3 install setuptools --user
apt-get install python3-matplotlib python3-psutil
```

B.2 Building and simulation

```
# install required Python packages
mkdir ~/ssdev
cd ~/ssdev
wget https://raw.githubusercontent.com/hewlettpackardlabs/supersim/master/...
chmod +x installpy
./installpy clone install

# install required C++ projects
git clone https://github.com/nicmcd/make-c-cpp ~/.makeccpp
cd ~/.makeccpp
make

# build required C++ programs
cd ~/ssdev
wget https://raw.githubusercontent.com/hewlettpackardlabs/supersim/master/...
chmod +x installcc
./installcc clone build

# generate settings JSON file for a simulation
using 'revised fat tree'
# SuperSim project as example
cp ../supersim/json/fattree_iq_blast.json sample.json

# run supersim with the generated settings
../supersim/bin/supersim sample.json \
   <DESIRED_MODIFICATION_OF_A_SETTING> ...
```